LE STRUTTURE E LE FUNZIONI AVANZATE.
PER SFRUTTARE AL MASSIMO IL PADRE DI TUTTI I LINGUAGGI

LAVORARE CON C++

Roberto Allegra

EDIZIONI MASTER

i libri di

LAVORARE CON

di Roberto Allegra

EDIZIONI MASTER

INDICE

Introduzione

1.1 Casting implicito 9
1.2 Casting esplicito "Vecchio Stile" 17
1.3 CONST_CAST 18
1.4 REINTERPRET_CAST 21
1.5 STATIC_CAST 27
1.6 Conclusioni 30

Gestione dinamica degli oggetti

2.1 Strumenti per l'analisi 34
2.2 SIZEOF 34
2.3 Struttura dei membri 36
2.4 Puntatori a membro 36
2.5 Puntatori a funzione e a metodo 39
2.6 Rappresentazione dei metodi 44
2.7 Rappresentazione dei metodi virtuali .47
2.8 VPTR E VTABLE 49
2.9 DYNAMIC_CAST 53
2.10 RTTI 55
2.11 Conclusioni 61

Eccezioni

3.1 La Classe Vettore 65
3.2 Sollevare un'eccezione 67
3.3 Gestire un'eccezione 70
3.4 Gestori Multipli 72
3.5 Gestore Generico 74
3.6 Gerarchie di classi eccezione 75
3.7 Specifica delle eccezioni 76
3.8 Gestori che rilanciano eccezioni 77
3.9 Eccezioni predefinite 78

3.10 Conclusioni 80

Paradigma Generico

4.1 Funzioni Template 83
4.2 Sovraccaricamento di funzioni template 88
4.3 Classi Template 89
4.4 Parametri multipli e costanti 91
4.5 Parametri predefiniti 93
4.6 Specializzazione dei Template 93
4.7 Conclusioni 94

Libreria e Contenitori Standard

5.1 Libreria C 97
5.2 Contenitori (Capitolo 5) 98
5.3 Algoritmi e Oggetti funzione (Capitolo 5) 99
5.4 Stringhe (Capitolo 6) 99
5.5 Stream (Capitolo 6) 100
5.6 Funzionalità matematiche 100
5.7 Supporto al linguaggio e diagnostica .101
5.8 Obiettivi dei contenitori C++ 102
5.9 Un esempio di contenitori: Vector 103
5.10 List e Iteratori 112
5.11 Deque e gli Adattatori 116
5.12 Contenitori Associativi 119
5.13 Punti critici nell’uso dei Contenitori .122
5.14 Algoritmi 126

Stringhe e Canali

6.1 CHAR_TRAITS 141
6.2 BASIC_STRING 142
6.3 Stream di Input e di Output 146
6.4 Stream predefiniti 150
6.5 Stringstream 150

INTRODUZIONE

Questo libro è il seguito del testo "Imparare C++", del quale amplia e completa il discorso.
Per capire questo volume è quindi necessario che tu abbia letto il predecessore, oppure che ne padroneggi i contenuti: i fondamenti del linguaggio, i tipi di dati, le strutture di selezione, la gestione di progetti articolati in più file e la definizione di classi e delle loro relazioni fondamentali (ad esempio: composizione, ereditarietà e polimorfismo).
Darò quindi per scontato che tu possegga questo "bagaglio base" di conoscenze grazie al quale è già possibile creare quasi ogni tipo di applicazione. Il C++, però, fornisce anche altri elementi più particolari e specifici, volti a tre finalità fondamentali: generare applicazioni più robuste, semplificare il lavoro del programmatore e superare vincoli formali o prestazionali. In questo libro, voglio appunto introdurre questi argomenti, la cui padronanza marca la vera distinzione fra coloro che scrivono codice aderente alle specifiche del linguaggio (il che non è sinonimo di "coloro che scrivono in C++": conosco gente che scrive in C++ pensando in Pascal, quindi in realtà scrive in Pascal) e quei fortunati che **usano** il C++ scegliendo accuratamente gli strumenti che permettano loro di rendere l'esperienza quanto più possibile comoda e **piacevole**. Enfatizzo la parola perché ad alcune persone può addirittura sembrare un'eresia, che lavorare in C++ sia piacevole. Spero che queste pagine servano proprio a chiarire questo concetto: il C++ è molto diverso dal C!

Entrando nel merito degli argomenti:

- **Il capitolo 1** tratterà nel dettaglio il **casting dei tipi**, introducendo gli operatori di conversione static_cast, reinterpret_cast, e const_cast..
- **Il capitolo 2** svelerà il "dietro le quinte" del C++, ovvero-

sia come un compilatore tipico organizza il codice e i riferimenti in memoria. Questo porterà alla spiegazione delle operazioni più potenti (e pericolose) permesse dal C++, con particolare riferimento alla **gestione dinamica degli oggetti** e all'operatore di conversione dynamic_cast..

- **Il capitolo 3** mostrerà come è possibile controllare in C++ le **eccezioni**, ovverosia quegli eventi imprevedibili che possono verificarsi durante il corso dell'esecuzione.
- **Il capitolo 4** illustrerà il **paradigma generico**: una potente arma che il C++ mette a disposizione per risolvere in modo elegante quelle situazioni in cui gli stessi algoritmi si applicano ad oggetti eterogenei.
- **Il capitolo 5** dipingerà un quadro panoramico della Libreria Standard del C++, e introdurrà i contenitori standard più usati. Sentirai parlare di vettori, liste, iteratori, algoritmi e oggetti funzione, e tanto altro..
- **Il capitolo 6** infine, concluderà il tutto con una leggera spiegazione di alcune funzionalità offerte dalle stringhe e dai canali standard.

Nonostante abbia cercato di essere quanto più scrupoloso possibile nella stesura di questo libro, molti argomenti specifici e approfondimenti non hanno potuto trovare posto.
Ad integrazione, ti suggerisco caldamente di studiare attentamente i testi riportati in Bibliografia, che costituiscono un insieme di riferimenti molto solido e completo per alla programmazione in C++.
Infine, ti consiglio, prima ancora di tuffarti nella lettura, di visitare il sito **www.robertoallegra.it**; nella sezione "libri" troverai gli errata corrige di questo testo e del suo predecessore, i codici sorgenti proposti, alcuni approfondimenti, nonché i modi per contattarmi in caso di difficoltà nel seguire il testo.
Alla fine della lettura dovresti avere acquisito le nozioni fondamentali per programmare in C++ in modo avanzato. Buona lettura!

CONVERSIONI

Una volta definita una serie di classi che hanno una qualche relazione fra loro, inizia progressivamente a sorgere l'esigenza di navigare attraverso la gerarchia che è stata creata. Tutti i modi in cui è possibile farlo prevedono uno o più **cast** (conversioni di tipo). In questo capitolo ci concentreremo proprio sui vari metodi che permettono di "passare da un oggetto all'altro".

1.1 CASTING IIMPLICITO

Nel primo libro abbiamo già visto che i cast spesso avvengono senza l'intervento del programmatore, quando una conversione ha un significato ovvio, non comporta perdita di informazione e non presenta ambiguità. Questo è immediato nel caso di conversioni di tipo primitivo che avvengano "per promozione":

```
int     i = 5;
long    l = i;    //bene: long >= int
char    c = l;    //warning! Perdita d'informazione
int*    p = i;    //errore! Nessun significato ovvio
```

Il penultimo assegnamento (c = l) comporta una perdita d'informazione, dal momento che un char ha un range di valori minore di un int: un compilatore sufficientemente pedante potrebbe rifiutarsi di proseguire, o più probabilmente generare un messaggio di avvertimento (warning). In ogni caso è necessario far capire al compilatore che siamo coscienti del rischio, attraverso un cast esplicito (vedi paragrafo 1.2).
L'ultimo assegnamento (p = i), invece, è proprio un errore: il cast, infatti, non ha alcun significato ovvio: come va interpretato "i" (vedi paragrafo 1.4)?
Quando si passa dal casting di dati primitivi a quello di oggetti valgono le stesse regole.

Ma come può essere ovvio un cast “da una classe all’altra”?

1.1.1 CONVERSIONE VIA COSTRUTTORE

PERMETTERE LA CONVERSIONE VIA COSTRUTTORE

Un primo modo di permettere il casting da un oggetto di tipo **B** ad uno di tipo **A** consiste nel realizzare un costruttore di A che presenti B come parametro, secondo lo schema:

```
class A
{
    A(B); //costruttore di A con parametro di tipo B
};
```

Abbiamo già visto un simile esempio nel primo libro, nella classe Frazione:

```
class Frazione
{
public:
    int numeratore;
    int denominatore;

    Frazione() : numeratore(0), denominatore(1) {};
    Frazione(int n) : numeratore(n), denominatore(1) {};
};

int main()
{
    Frazione f;
    f = 1;          // giusto! f = Frazione(1)
    return 0;
}
```

Per rendere più evidente il passaggio qui ho presentato due costruttori (e non uno solo con parametri predefiniti). Il cast "Frazione = int" nell'istruzione f=1 è corretto, perché il compilatore può costruire efficacemente un costruttore in Frazione che accetti un int come parametro. In questo caso la conversione implicita ha un valore equivalente a: **f = Frazione(1).**

IMPEDIRE LA CONVERSIONE IMPLICITA VIA COSTRUTTORE

La conversione implicita via costruttore è spesso comoda, perché rende la programmazione più intuitiva. Il rischio, però, è che il compilatore segua le regole interpretando come "ovvie" conversioni che invece sono ambigue e si prestano a facili fraintendimenti.
L'esempio tipico è quello della classe string, che può essere costruita sia indicando una stringa letterale, sia indicando il numero dei caratteri che ne faranno parte:

```
string str1("Ciao");         //stringa "Ciao"
string str2(5);         //stringa di 5 elementi
```

Il problema è che molti programmatori tendono distrattamente a confondere questo comportamento, aspettandosi una conversione implicita da int a string (che non esiste, dal momento che esiste il costruttore numerico), o ancor di più da char a string.
Un programmatore distratto potrebbe scrivere qualcosa del genere:

```
string str1 = 1984;         //Questa stringa contiene 1984 caratteri!
string str2 = '@';       //Questa stringa contiene 64 caratteri!
```

Chi si aspetta che str1 contenga il titolo di un romanzo di Orwell, e str2 una chiocciolina rimarrà deluso: str1 è una stringa di 1984 caratteri, e str2 è una stringa di '@' (ovvero 64) caratteri.
Ciò avviene perché il compilatore ha seguito diligentemente le regole,

dando per implicite queste conversioni:

```
string str1 = string(1984)
string str2 = string(int('@'))
```

Stabilire in anticipo quando possono verificarsi queste conversioni indesiderate spetta al progettista della classe, che dispone di uno strumento apposito per impedirle: la parola chiave explicit.
Questa permette di indicare un costruttore che dovrà per forza essere richiamato esplicitamente, pena un errore di compilazione. Una verisione semplificata di string (che non corrisponde a quella reale, come vedremo nel capitolo 7, ma che centra il punto), potrebbe essere questa:

```
class string {
    string(const char *str) {…}       //stringa in stile C
    explicit string(int g) {…}        //grandezza della stringa
};
```

Poiché il costruttore di string(int) è dichiarato come esplicito, gli assegnamenti erronei di prima saranno rifiutati dal compilatore.
Nel caso in cui si volesse perseguire realmente lo scopo di dichiarare un array di 1984 caratteri sarà comunque possibile usare le scritture esplicite:

```
string str1 = 1984;          //Errore: deve essere richiamato esplicitamente
string str(1984);        //Giusto
string str = string(1984);   //Giusto
```

1.1.2 OPERATORE DI CONVERSIONE

L'operazione complementare alla definizione di un costruttore parametrico è quella in cui si ha un oggetto di tipo **A** e si vuole specificare come, a partire da esso, sia possibile costruire un oggetto di ti-

po **B**: ciò può essere facilmente realizzato attraverso la definizione di uno o più **operatori di conversione**. Dallo schema che segue puoi evincere la sintassi della dichiarazione di un operatore di conversione:

```
class A
{
    operator B() {codice};
};
```

L'esempio seguente permette di ottenere un valore double a partire da una Frazione:

```
class Frazione {
public:
    int numeratore;
    int denominatore;
    Frazione(int n, int d) : numeratore(n), denominatore(d) {}
operator double() {
        return    (double)numeratore /
            (double)denominatore;
    }
};
int main()
{
    Frazione f(20,8);
    double d = f;  //giusto: 2.5
    return 0;
}
```

1.1.3 AMBIGUITÀ

I due sistemi appena descritti possono rendere la programmazione naturale e coerente, a patto che si progettino le conversioni in ma-

niera ragionata. Ad esempio, tenendo buona la definizione di Frazione del paragrafo precedente, è possibile scrivere un programma del genere:

```
#include <iostream>
using namespace std;

//definizione di Frazione

int main()
{
    Frazione f(20,8);
    cout << f;
    return 0;
}
```

in questo caso l'operazione di cout darà buon fine, anche se non è stato definito un operatore di inserimento specifico per il tipo Frazione. In mancanza di una definizione come:

```
operator<<(ostream&, const Frazione&)
```

il compilatore ha cercato una corrispondenza diversa, assumendo un cast implicito in double, richiamando:

```
operator<<(ostream&, const double)
```

il cui comportamento è già definito dal linguaggio. L'output viene quindi generato automaticamente:

```
2.5
```

Capire come il compilatore interpreta le chiamate, come prova a tro-

vare corrispondenze assumendo cast impliciti, è quindi fondamentale per comprendere il corretto funzionamento dei programmi, e anche per prevedere quando il compilatore potrà trovarsi in difficoltà. Ad esempio:

```
class Frazione
{
    //…
    operator double() {return         (double)numeratore /
                          (double)denominatore;}
    operator int()
{return numeratore / denominatore;}
};

int main()
{
    Frazione f(20,8);
    cout << f;            //quale operatore uso???
    return 0;
}
```

In questo caso ho definito due operatori di conversione: uno per **int** e uno per **double**. L'effetto collaterale è che ora il compilatore non ha più modo di stabilire quale utilizzare per convertire l'oggetto **f**. In simili situazioni di **ambiguità** verrà generato un errore e verrà richiesto di specificare l'operatore desiderato attraverso un cast esplicito.

1.1.4 UPCASTING

Un'altra conversione tanto sicura da essere assunta come implicita è quella che prevede un passaggio da un oggetto di tipo derivato ad uno di tipo base. Introduciamo la semplice gerarchia, riassunta dalla figura 1.1
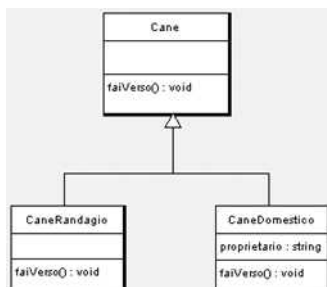
Figura 1.1: Semplice gerarchia di classi

```
class Cane {
public:
    void faiVerso() {cout << "Bau!";}
};

class CaneRandagio : public Cane {
public:
    void faiVerso() {cout << "Grrr!";}
};

class CaneDomestico : public Cane {
public:
    string    padrone;
    void faiVerso() {cout << "Arf!";}
};
```

Una conversione che avvenisse da CaneDomestico (o CaneRandagio) a Cane sarebbe data per implicita.

```
int main()
{
    CaneDomestico pluto;
```

```
      pluto.padrone = "Topolino";
      Cane& caneGenerico = pluto; //upcast implicito
      return 0;
}
```

Una conversione del genere viene detta upcast (cast all'insù), perché il riferimento "risale" la gerarchia presentata in figura.
L'upcasting permette di trattare in maniera generica classi specializzate, ma appiattisce le differenze fra i vari oggetti: dopo di esso, infatti, non sarà più possibile accedere alla parte d'informazione definita nella classe derivata, ma solo ai membri e alle funzioni definiti nella classe base.

1.2 CASTING ESPLICITO "VECCHIO STILE"

Nel libro "Imparare C++" ho già introdotto un tipo generico di casting esplicito, mutuato direttamente dal C, che si utilizza indicando il tipo verso il quale si vuole effettuare la conversione all'interno delle parentesi tonde. Degli esempi di questo genere di casting sono riportati frequentemente anche nei paragrafi precedenti.
Allo stesso modo è anche possibile utilizzare il vecchio casting "in stile funzionale", trattando la conversione come fosse una normale funzione. Prendendo ad esempio il codice riportato nel paragrafo precedente:

```
cout << (double)f;  //cast C-like
cout << double(f);  //cast funzionale
```

È indifferente scegliere una delle due forme, sebbene la prima sia quella più adottata comunemente, dal momento che è possibile usarla anche con tipi che prevedano puntatori e riferimenti. Questo tipo di casting è una specie di passepartout: indica al compilatore di for-

zare una conversione di tipo e provare in diverse maniere a soddisfare la richiesta. Questo punto va ribadito ulteriormente: il compilatore, in realtà, traduce il casting in stile C in molti modi diversi (che vedremo), a seconda delle circostanze in cui si trova.
Dal momento che i programmatori non stanno mai molto attenti alle sfumature semantiche delle operazioni che scrivono, e poiché il casting è una cosa molto seria ed è rischioso interpretarlo in maniera scorretta (da ambo le parti), il C++ prevede una serie di operatori di conversione che ne specificano l'esatto significato in modo inequivocabile: questi sono **static_cast**, **const_cast**, **reinterpret_cast** e **dynamic_cast**.

1.3 CONST_CAST

Per farti capire esattamente cosa intendo per "sfumature diverse di significato" nella conversione, comincio l'analisi degli operatori di casting da **const_cast**, la cui sintassi è:

```
const_cast<NuovoTipo>(espressione)
```

Const_cast si usa per rimuovere il vincolo di costanza di un puntatore o di un riferimento. Immagina, ad esempio, questa situazione:

```
class Frazione
{
    //definizione di Frazione
    bool visualizzata;
};
void StampaFrazione(const Frazione& f)
{
    f.visualizzata = true; //errore: f è const!
    cout << f;
};
```

In questo caso la classe Frazione definisce un campo booleano visualizzata per capire se è stata stampata almeno una volta.
StampaFrazione, invece, accetta un parametro per riferimento costante, e (per un qualsiasi motivo) non ci è dato di ridefinirne il prototipo. Il risultato è che il compilatore non accetterà il nostro assegnamento a f.visualizzata, perché viola apertamente il vincolo di costanza. L'unico metodo per effettuare quest'operazione con gli operatori di casting è usare **const_cast**:

```
Frazione& fNonCostante = const_cast<Frazione&>(f);
fNonCostante.visualizzata = true;
```

In questo modo ho dichiarato un riferimento normale (non costante) e ho utilizzato l'operatore const_cast per togliere il vincolo di costanza: in altre parole ho forzato una conversione da (const Frazione&) a (Frazione&).
C'è un bel numero di obiezioni che potresti pormi su quest'operazione: cercherò di immaginare le principali e darti una risposta.

Obiezione 1: Ma hai barato! Se qualcuno ha definito quel vincolo di costanza ci sarà stato un buon perché.

Hai ragione. In effetti le operazioni di const_cast sono pericolose perché permettono di minare alla base ogni buon impianto architetturale, aggirando bellamente i vincoli imposti da design.
Tanto più che il C++ prevede moltissimi strumenti molto più robusti di un const_cast: ad esempio, sarebbe stato possibile definire visualizzata come membro mutable. D'altra parte esistono rari casi in cui un const_cast si presenta come la soluzione più semplice ad un giro di modifiche che si rivelerebbe alrimenti troppo oneroso

Obiezione 2: Ma non posso fare la stessa cosa con un casting in stile C?

La risposta è sì: il codice seguente è perfettamente legale, anche se un compilatore pedante potrebbe generare un messaggio di avvertimento.

```
Frazione& fNonCostante = (Frazione&)(f);
fNonCostante.visualizzata = true;
```

Obiezione 2/bis: Ma allora a che serve usare l'operatore di const_cast?

Se ti stai ponendo questa domanda, non sono riuscito ancora a comunicarti lo spirito con cui si usano gli operatori di casting in C++: attraverso una conversione in stile C si può forzare una conversione di (quasi) ogni tipo, e proprio questo fatto sta all'origine di molti bug. Rileggi la prima obiezione, ripensa a quanto è infido l'uso di un casting di questo tipo, e poi guarda con quanta nonchalance siamo riusciti ad infilarlo nel codice dell'obiezione 2.
Dichiarare esplicitamente il casting attraverso un const_cast equivale a dire (a chi legge e, soprattutto, al compilatore): "so quello che sto facendo, so che non è bello, ma è esattamente il comportamento che voglio ottenere".
Con un casting in stile C diventa impossibile riconoscere questo particolare inganno da altre conversioni molto più innocue. È proprio questo che intendo quando parlo di "specificare e distinguere le sfumature semantiche di un cast".

Obiezione 3: Il codice d'esempio presuppone dei vincoli irreali. In una situazione vera basterebbe togliere il modificatore 'const' dal parametro.

Indubbiamente, ma gli esempi "semplificano" per definizione.
Esistono scenari appena più complicati che possono porre dei veri problemi: pensa ad una situazione in cui si derivi da una classe presen-

te in una libreria, e occorra ridefinirne un metodo. In questo caso non è possibile ritoccare il prototipo della funzione.
Un'applicazione ancora più realistica di const_cast svela il famigerato trucco per aggirare addirittura l'intero vincolo di costanza logica di una funzione:

```
class A
{
    string intoccabile;
    void funzioneCostante() const
{
        A* thisNonCostante = const_cast<A*>this;
        thisNonCostante->intoccabile = "Toccata!";
    }
};
```

Un bel const_cast dal tipo di this (const A*) ad un semplice A*… e addio a tutti i sogni di robustezza del design! Se si usa const_cast in queste situazioni, invece delle conversioni in stile C, sarà sempre possibile andare a verificare se sono stati adottati simili pericolosi escamotage, semplicemente con una funzione di ricerca testuale.

1.4 REINTERPRET_CAST

L'operatore di conversione reinterpret_cast è di gran lunga il più pericoloso fra quelli previsti dal C++, e normalmente si usa molto poco se la programmazione non scende a basso livello.
Proprio per questo, sarà bene cominciare questa dissertazione con un piccolo anticipo di ciò che approfondiremo nel capitolo 2: come gli oggetti vengono normalmente rappresentati in memoria, durante l'esecuzione del programma.
Impareremo qualcosa sui sistemi a basso livello ,ma sicuramente ci tornerà molto utile in futuro.

1.4.1 RAPPRESENTAZIONE DEGLI OGGETTI IN MEMORIA

Facciamo un esperimento: prendiamo il compilatore Visual C++, che come molti altri rappresenta un int con 4 byte.
Quindi scriviamo questo codice:

```
class Frazione
{
public:
    int numeratore;
    int denominatore;

    Frazione(int n, int d) : numeratore(n), denominatore(d) {};
    //definizione dei metodi [...]
};

int main()
{
    Frazione f(8, 6);
    return 0;
}
```

A questo punto ti pongo una domanda indiscreta: "in che modo il compilatore rappresenta **f** in memoria?".
"Saranno affari del compilatore", risponderai tu, e normalmente avresti anche ragione.
Ma saper rispondere, in questo caso, è fondamentale per capire il funzionamento dell'operatore **reinterpret_cast**.
In figura 1.2 puoi vedere l'informazione recuperata direttamente dalla finestra di debug di quest'ottimo IDE.
Questa ci rivela l'indirizzo al quale è memorizzata la variabile f, e offre una fotografia della porzione di stack in cui essa si trova; in particolare, la zona marchiata in negativo è quella occupata dalla variabile.

Ci vuole poco a capire che l’ambiente pone sullo stack soltanto i valori dei membri, nell’ordine rovesciato tipico delle architetture little-endian.
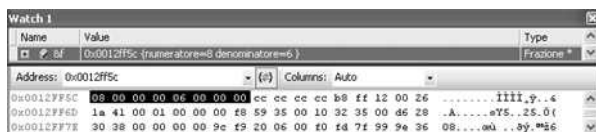
Figura 1.2: Informazioni di debug sulla variabile Frazione f(8,6)

1.4.2 REINTERPRETAZIONE DI PUNTATORI

L’operatore reinterpret_cast serve a fornire un’altra interpretazione al blocco di memoria detenuto dalla variabile; ad esempio potremmo dare al compilatore un ordine del genere: “so bene che la variabile f è una Frazione, ma da adesso voglio che la tratti come se fosse un int”.
Questo si traduce nel codice:

```
int main()
{
    Frazione f(8, 6);
    int& i = reinterpret_cast<int&>(f);
    cout << i;
    return 0;
}
```

Quale sarà l’output generato da questo codice?
Guarda nuovamente la figura, e la risposta sarà semplice:

```
8
```

Abbiamo infatti definito un nuovo riferimento di tipo int, che punta

al primo byte detenuto da f.
Poiché in questo compilatore un int prende quattro bytes, i viene a coincidere con il valore di **numeratore**.
È quindi possibile utilizzare reinterpret_cast per far puntare un dato di un tipo qualsiasi da un puntatore di un altro tipo qualsiasi.
Non ci sono parole per esprimere quanto una simile operazione sia potenzialmente pericolosa e perfino priva di senso se viene eseguita alla leggera: se avessi tentato di reinterpretare f come un tipo di grandezza maggiore di 8 byte, ad esempio, questo avrebbe sconfinato in zone di memoria non utilizzate e/o protette.
Risultato: crash alla prima operazione sull'oggetto.

1.4.3 REINTERPRETAZIONE VALORE -> INDIRIZZO

Un'operazione intimamente simmetrica a quella appena vista è la trasformazione di un'espressione in un indirizzo.
Ti propongo questo piccolo "analizzatore di memoria", che richiede all'utente un indirizzo e restituisce il byte in esso contenuto.

```
int main()
{
    while(true)
{

        cout << "Inserisci l'indirizzo che vuoi leggere: ";
        int indirizzo;
        cin >> indirizzo;

        char* cella = reinterpret_cast<char*>(indirizzo);
        cout << "Il valore contenuto e': " << (int)*cella;
    };
    return 0;
}
```

Qui la reinterpretazione ha luogo fra l'indirizzo indicato come intero e un puntatore a char. In questo modo è possibile puntare un singolo byte, del quale sarà poi stampato il valore.

1.4.4 CASTING VERSO VOID*

L'uso di reinterpret_cast descritto nel paragrafo 1.4.2 può essere riassunto come "eliminare il vincolo di tipizzazione da un puntatore". Questo significa che ci si può addirittura disinteressare completamente del tipo del puntatore, dal momento che si può comunque ripristinarlo in seguito; un puntatore di questo tipo può essere efficacemente rappresentato dal tipo void*.

```
Frazione f(8,6);
void* v = reinterpret_cast<void*>(&f);
```

Un puntatore void* è la memorizzazione di un "indirizzo puro", senza alcuna informazione di tipo.
Da ciò deriva necessariamente il fatto che una variabile void* non potrà mai essere dereferenziata.

```
cout << *v; //errore: v non ha tipo.
```

Per riuscire ad utilizzare efficacemente il contenuto di v sarà necessario ripetere un reinterpret_cast inverso che riporti il tipo da void* a Frazione*.

1.4.5 TYPE PUNNING

A questo punto forse ti starai chiedendo a che serve utilizzare il tipo void*, dal momento che sembra utile solo per compiere "giri a vuoto". Se è così, direi che è giunta l'ora di introdurre uno dei problemi con cui ci confronteremo per parte del libro: creare dei contenitori "versatili", in grado di includere oggetti di tipo eterogeneo. Immaginiamo, ad esempio, di disporre di una serie di oggetti, di natura del tut-

to scorrelata: un **Aereoplano a**, un **Bastone b** ed una **Casa c**. Ora, vogliamo inserirli in un array: dato che in C++ i vettori sono tipizzati, l'unica possibilità è proprio quella di dichiarare un array di void*.

```
void* oggetti[] = {     reinterpret_cast<void*>(&a),
            reinterpret_cast<void*>(&b),
            reinterpret_cast<void*>(&c)};
```

Ecco spiegata l'utilità dei puntatori di tipo void*: certo, a ben pensarci avremmo potuto dichiarare oggetti[] come un puntatore di un qualunque tipo, e sarebbe andato bene comunque, ma **void*** indica precisamente che nell'array sono contenuti oggetti di un non-tipo, e ci assicura che nessuno tenterà di dereferenziare la variabile senza aver prima provveduto alla necessaria conversione. Questo genere di trattamento viene definito **type punning**, è concettualmente analogo all'uso delle **union**, ed è considerato (a ragione) una pratica molto insicura. Uno dei problemi fondamentali è che una volta ottenuto l'array non c'è proprio alcun'informazione valida che ci permetta di stabilire il tipo originale dei vari elementi. Una soluzione può essere quella di creare una nuova struttura che memorizzi anche un'informazione di tipo:

```
enum TipoOggetto
{
    AEREOPLANO,
    BASTONE,
    CASA
};
struct Oggetto
{
    TipoOggetto tipo;
    void* puntatore;
}
```

Ciò permette di eseguire quantomeno una verifica di tipo per scegliere il casting opportuno ed evitare conversioni errate.
D'altra parte, in questo modo si forza l'uso di strutture necessariamente note a priori che devono essere enumerate in TipoOggetto.
Così facendo, l'analogia con le union si rafforza (Oggetto somiglia pericolosamente al tipo Variant, che i programmatori Visual Basic conoscono bene, ed evitano accuratamente), e diventa spontaneo chiedersi perché non preferire l'uso di soluzioni più comode e sicure, come la definizione di una gerarchia di classi dal comportamento polimorfico.

1.5 STATIC_CAST

Probabilmente, al punto in cui siamo giunti, la modalità più semplice di definire **static_cast** è quella "per esclusione": se un'operazione si può eseguire con un casting in stile C, e questa non ricade nei casi previsti da const_cast e reinterpret_cast, allora si può fare con static_cast.
La definizione appena data è immediata e sostanzialmente corretta (c'è un'importante eccezione che vedremo nel prossimo capitolo), ma poco nitida dal punto di vista formale.
Precisando un po', quindi, static_cast può essere usato per tutte le conversioni di tipo implicito (primitive, per operatore di conversione, costruttore e upcast).
Oltre a queste, ricadono nella sua sfera di competenza alcune conversioni triviali come l'aggiunta di costanza o volatilità a un tipo, la trasformazione di tipi interi in enumerazioni, e l'inverso di alcune conversioni implicite (ad esempio, da int a char, con tutti i rischi di perdita di informazione del caso). Una particolare conseguenza di quest'ultima affermazione è che attraverso static_cast è possibile realizzare anche l'inverso di un upcast, ovverosia una conversione da un tipo base ad un tipo derivato. Un'operazione del genere viene detta **downcast**, e merita un paragrafo a parte.

1.5.1 DOWNCASTING

Riprendiamo la situazione illustrata in figura 1.1, e proviamo a definire il comportamento di un Accalappiacani.
Il lavoro di costui è prendere un cane e portarlo al canile se è randagio, o restituirlo al legittimo proprietario se è domestico.
Possiamo dichiarare la sua classe così:

```
class Accalappiacani
{
public:
    void recupera(Cane& cane);
};
```

Uno dei problemi fondamentali dell'Accalappiacani è che si trova davanti un'istanza troppo generica di Cane, dalla quale non è possibile evincere nulla della classe derivata.
Ad esempio, potremmo invocare il suo aiuto in un caso simile:

```
int main()
{
    CaneDomestico pluto;
    pluto.proprietario = "Topolino";
    Accalappiacani tizio;
    tizio.recupera(pluto);
    return 0;
}
```

In questo caso, è stata passata un'istanza di CaneDomestico, ma l'Accalappiacani si ritrova a gestire solo l'informazione riguardante un cane generico, pertanto non può risalire direttamente al suo padrone.
Per riuscire nell'impresa, un Accalappiacani astuto deve aggirare l'ostacolo forzando un **downcast**:

```
void Accalappiacani::recupera(Cane& cane)
{
    //downcast da Cane& a CaneDomestico&
    CaneDomestico& dCane =  static_cast<CaneDomestico&>(cane);

    //ora posso accedere al membro "padrone"
    //in modo efficace
    cout<< "vado a riportare il cane a "
        << dCane.padrone;
}
```

L'output dell'applicazione non ci riserverà sorprese:

```
vado a riportare il cane a Topolino
```

In questo modo abbiamo risolto, per mezzo di un downcast, il problema di accedere ad un membro della classe derivata.
Ci siamo riusciti perché il cane in questione era **realmente** un cane domestico, pertanto l'operazione è stata valida. Ma questo è qualcosa che non possiamo dare per scontato, né possiamo sempre sapere in anticipo.
Proviamo a sottoporre al nostro Accalappiacani troppo ottimista questa situazione:

```
int main()
{
    CaneRandagio ringhio;
    Accalappiacani tizio;
    tizio.recupera(ringhio);
    return 0;
}
```

L'output di un'esecuzione sulla mia macchina è stato:

```
Vado a riportare il cane a Kf@°_ô
```

La ragione di questo disastro è evidente: abbiamo sbagliato il cast. Il cane non era affatto domestico, quindi non aveva un padrone, quindi il cielo soltanto sa a cosa abbiamo puntato nell'istruzione cout. Il comportamento in questo caso è ancor più pericoloso e imprevedibile, perché static_cast non dà alcun segno quando l'operazione non va a buon fine, e per un motivo preciso: non può saperlo. Il C++ è un linguaggio tipizzato staticamente, e tutte le risoluzioni avvengono al momento della compilazione.
E al momento della compilazione non c'è nulla che possa far presagire se una particolare chiamata di Accalappiacani::recupera() riguarderà un randagio o meno.
Per ottimizzare le prestazioni, inoltre, il C++ non memorizza alcuna informazione riguardo al tipo all'interno della variabile (l'abbiamo visto in 1-4, e lo vedremo meglio nel prossimo capitolo).
In questo modo non c'è maniera di riconoscere la legalità dell'operazione né a tempo di compilazione né a runtime.
Tizio, per questa volta, deve quindi accettare la sconfitta: è invariabilmente destinato a fallire nel 50 percento dei casi.
Ma è una persona intraprendente, e lo reincontreremo presto.

1.6 CONCLUSIONI

Il casting è uno strumento potente e spesso necessario.
Il C++ permette di tenere d'occhio le diverse sfumature che i cast possono assumere attraverso gli operatori di conversione.
Il mio consiglio è di usarli sempre, preferendoli alla scrittura in stile C, che può sempre essere riproposta come uno static_cast, const_cast, reinterpret_cast o dynamic_cast (spiegherò quest'ultimo nel prossimo capitolo).
Un altro consiglio è di non abusare di certi tipi di cast: il reinterpret_cast, per esempio, è particolarmente subdolo è incline a bug

di vario tipo. Anche operazioni allettanti come il type punning dovrebbero sempre essere evitate.
Non vale proprio la pena di complicarsi l'esistenza con trucchetti del genere: nel corso di questo libro introdurremo dei concetti che, uniti ad un buon design dell'applicazione, elimineranno ogni necessità del ricorso a soluzioni così intrinsecamente pericolose.
Anche il downcasting con static_cast è da vedere con sospetto, soprattutto se non si ha la completa certezza di stare eseguendo realmente un cast verso l'interfaccia corretta.0

GESTIONE DINAMICA DEGLI OGGETTI

Il C++ è un linguaggio **tipizzato staticamente**.
Come sappiamo, questo vuol dire che tutte le decisioni riguardanti classi, oggetti, e casting vengono prese al momento della compilazione, di modo che sia sempre determinato con certezza assoluta il tipo delle variabili utilizzate nel programma, indipendentemente da ciò che avverrà nel corso dell'esecuzione. In realtà abbiamo visto che questo non è sempre fattibile: per alcuni tipi di operazioni, come il type punning, il casting da void* e il downcasting, è impossibile stabilire a priori il tipo delle variabili in gioco. In questo caso, per essere più precisi, le variabili **hanno** un tipo (in C++ gli oggetti sono sempre tipizzati), ma questo è stato alterato tramite casting ed è diventato completamente irrilevante (ad esempio, void*) o troppo generico (ad esempio, Cane&).
Il C++ permette un certo grado di gestione dinamica della programmazione, e in alcuni casi è possibile rintracciare il tipo originale di alcuni oggetti.
Un libro può spiegare come e perché questo sia realizzabile, in due modi: trattando i casi in maniera semplice ma apodittica, oppure analizzando ciò che avviene dietro le scene, e come opera un compilatore. Il rischio, in questo secondo caso, sta nel fatto che più si scende nel dettaglio, più la trattazione diventa monotona, noiosa, e fine a se stessa.
Per questo, nello scegliere quest'ultima via, mi sono limitato alla spiegazione dei fatti essenziali per la comprensione della gestione dinamica degli oggetti (se sei davvero interessato a come funziona un compilatore, ti consiglio di leggere [11] - si tratta comunque di un'esperienza interessante e che migliora la comprensione della programmazione); spero di aver combinato così semplicità e senso critico. Ed ora prepariamoci per il nostro viaggio "under the hood"!

2.1 STRUMENTI PER L'ANALISI

Voglio innanzitutto puntualizzare che i compilatori in commercio sono implementati nelle maniere più disparate: lo standard C++ non forza in alcun modo la struttura interna di un compilatore, ma solo il comportamento che esso deve assumere nei confronti del linguaggio.
Tuttavia le funzionalità che esporrò qui sono utilizzate più o meno da tutti i compilatori senza grosse variazioni.
Per l'analisi del "dietro le quinte" mi servirò, come ho fatto nel paragrafo 1.4, di Visual C++ e del suo IDE, che propone un debugger molto intuitivo e potente - se vuoi fare delle prove pratiche da te, basta un qualsiasi debugger che permetta il watching delle variabili e l'analisi della memoria.
Alcune informazioni statiche, però, possono anche essere facilmente desunte operando "dall'interno del linguaggio", ad esempio usando **sizeof**.

2.2 SIZEOF

Sizeof è un operatore che restituisce lo spazio occupato da una variabile o da un tipo, espresso in char.
Quindi, dal momento che tipicamente un char corrisponde a un byte, possiamo dire che sizeof rivela quanti byte "prende" un dato o un tipo. Lo standard prevede ben due sintassi diverse, a seconda che si stia analizzando un'espressione o un tipo di dati:

```
sizeof espressione
sizeof(tipo dati)
```

Nella pratica della maggioranza, comunque, viene aggiunta all'espressione una coppia di parentesi, cosicché si possa usare sempre una sintassi coerente con quella del secondo tipo.
Esempi dell'utilizzo di sizeof sono:

```
int a;
sizeof(int)    //minimo 2, probabilmente 4
sizeof(a)      //lo stesso risultato di sizeof(int)
```

Nel caso l'espressione sia un array, sizeof restituisce la dimensione totale occupata dagli elementi in esso contenuti:

```
char stringa[] = "CIAO";   //5 caratteri (c'è un carattere nullo alla fine)
sizeof(stringa)            //sicuramente 5
```

Sizeof, comunque, è un operatore statico e non acquisisce informazioni a tempo di esecuzione.
Gli è impossibile, ad esempio, stabilire quanto sia grande un array, se questo viene passato come parametro di una funzione:

```
int lunghezzaStringa(char stringa[]) {
    return sizeof(stringa) - 1;
}
```

Si potrebbe pensare, dopo aver scritto il codice riportato qui sopra, di aver trovato una funzione magica che valuti la lunghezza massima del buffer di una stringa senza dover esplicitamente memorizzare alcun valore costante.
In realtà il compilatore non può avere alcuna informazione su una stringa che sarà passata solo a tempo di esecuzione, per cui in un simile caso sizeof si limiterà a restituire la dimensione occupata dal **puntatore** *stringa (e i puntatori di solito prendono 4 byte).
Il che vuol dire che lunghezzaStringa, purtroppo, darà sempre 3.
Quando vengono analizzate espressioni di tipo composto (ad esempio, classi o struct), sizeof può rivelare delle belle sorprese, e introdurre un volenteroso neofita a scoperte nuove: dai **padding bytes** (byte inutilizzati che vengono aggiunti a quelli dei membri per allineare la memoria a margini prefissati) ai **puntatori vir-**

tuali (che vedremo presto).

2.3 STRUTTURA DEI MEMBRI

Abbiamo già visto che agli oggetti viene riservato uno spazio in memoria nel quale vengono allocati i vari membri, secondo l'ordine in cui vengono presentati all'interno della classe.
Prendiamo ad esempio la classe Frazione presentata nel paragrafo 1.4. La figura 1.2 mostra chiaramente la disposizione dei membri.
Ora, poniamo di scrivere un codice simile:

```
int main()
{
    Frazione f(8,6);
    cout << f.numeratore;
    cout << f.denominatore;
}
```

Vediamo i riferimenti di questo semplice codice dal punto di vista del compilatore: per memorizzare f sarà allocato, in un dato punto dello stack (poniamo 0x0012FF5C, sempre seguendo la figura 1.2), uno spazio di dimensione pari alla somma di quella dei membri (numeratore + denominatore = int + int = 4 + 4 = 8).
I membri di un oggetto, quindi, potranno essere individati semplicemente attraverso degli **spiazzamenti** all'interno di tale blocco.
Nel nostro esempio, numeratore può essere indicato come "&f + 0" e denominatore "&f + 4".

2.4 PUNTATORI A MEMBRO

Ora che sai quale segreto si cela dietro i membri, puoi capire bene una delle caratteristiche meno note del C++: i **puntatori a membro**.
Si tratta di puntatori particolari, che godono di diverse proprietà e pos-

sono essere utilizzati in molti modi.
Innanzitutto si dichiarano come dei puntatori normali: hanno un tipo, un nome, e sono indicati con il solito asterisco al quale, però, si prepone la classe d'appartenenza seguita dall'operatore di "risoluzione di visibilità" (::).
Ad esempio:

```
int Frazione::* membro;
```

inizializza il puntatore "membro" che potrà solo essere utilizzato per puntare ad un membro intero della classe Frazione:

```
int Frazione::* membro = &Frazione::denominatore;
```

A questo punto il puntatore membro conterrà l'esatto spiazzamento da aggiungere ad una variabile di tipo Frazione per fare riferimento al membro "denominatore".
Quindi il suo valore sarà pari a 4 byte (ovverosia 1 int: lo spazio occupato da "numeratore").

```
int main()
{
    int Frazione::*membro = &Frazione::denominatore;
    cout << membro;  //da quanti int è composto lo spiazzamento?
    return 0;
}
```

Eseguendo questo codice avremo una risposta coerente:

```
1
```

Difficilmente un puntatore a membro viene utilizzato per avere informazioni sullo spiazzamento di un attributo.

Più comunemente lo si applica all'istanza di un oggetto, come se fosse un alias del membro al quale punta.
Ad esempio:

```
int main()
{
    int Frazione::* membro = &Frazione::denominatore;
    Frazione a(8,6);
    cout << a.*membro;
}
```

```
6
```

Usando a.*membro diciamo al compilatore: "prendi quell'attributo di a che sta allo spiazzamento indicato dal puntatore "membro"". Cioè denominatore, cioè (in questo caso) 6.
Pochi programmatori C++ ricorrono ai puntatori a membro, fondamentalmente perché le occasioni di farlo si restringono a non più di un paio di pattern formali; tuttavia è utile sapere che esistono.
Con un po' di fantasia è facile trovare un esempio in cui i puntatori a membro si rivelano una scelta flessibile e immediata:

```
class Persona
{
public:
    string nome;
    string cognome;
    Persona(string n, string c) : nome(n), cognome(c) {};
};

int main()
{
    Persona persona[] = {Persona("Tizio", "Tizi"),
```

```
                Persona("Caio",  "Cai"),
                Persona("Sam", "Pronyo")};

    char risposta = '\0';
    while(risposta != 'N' && risposta != 'C') {
        cout << "Vuoi la lista dei nomi[N] o dei cognomi[C]? ";
        cin >> risposta;
    };

    string Persona::* membro = (risposta == 'N' ? &Persona::nome :
                    &Persona::cognome);
    for (int i=0; i<3; i++)
        cout << persona[i].*membro << endl;
    return 0;
}
```

Come vedi, l'uso più tipico di questo tipo di puntatori consiste nella "selezione" del membro di una classe, a run time (in questo caso la scelta viene fatta dall'utente).
Quando questo processo non si esaurisce nel giro di poche righe come in questo semplice esempio, ma dev'essere fatto perdurare nel corso dell'applicazione, i puntatori a membro regalano un fondamentale contributo alla gestione dinamica degli oggetti.

2.5 PUNTATORI A FUNZIONE E A METODO

Il concetto dei "puntatori a membro" si può applicare in maniera analoga anche ai metodi.
È cioè possibile realizzare un puntatore che faccia riferimento un metodo della classe. Io chiamo questo genere di puntatori "Puntatori a metodo", altri preferiscono la dicitura "Puntatori a funzione membro".

Questo genere di puntatori è intimamente connesso ai puntatori a funzione, che il C++ mutua direttamente dal C; pertanto analizzeremo questi per primi.

2.5.1 PUNTATORI A FUNZIONE

Un puntatore a funzione può fare riferimento ad una funzione qualsiasi, una volta che sia stato stabilito il tipo di valore che questa dovrà restituire e gli argomenti che dovrà esporre.
Una dichiarazione di puntatore a funzione, quindi, sarà molto simile a quella di un prototipo, a parte il fatto che il nome sarà preceduto da un asterisco e scritto fra parentesi.
Ad esempio, il codice:

```
double (*puntatore) (int, int);
```

Dichiara un puntatore a funzione di nome "puntatore", che può agganciarsi a funzioni che restituiscano un valore double e richiedano due argomenti int, come queste:

```
double dividi (int a, int b) {return double(a)/double(b);}
double moltiplica (int a, int b) {return a*b;}
double somma (int a, int b) {return a+b;};
double sottrai (int a, int b) {return a-b;};
```

La referenziazione viene effettuata in maniera molto semplice, trattando la funzione da puntare come se fosse una variabile.
Ad esempio, date le precedenti definizioni, questo codice associa puntatore alla funzione moltiplica:

```
puntatore = &moltiplica;
```

Una volta associato ad una funzione, il puntatore può essere utilizzato per richiamarla, dereferenziandolo.

Ovviamente, occorrerà anche passare gli eventuali argomenti:

```
(*puntatore) (4,3);
```

La riga scritta qui sopra restituirà 12.0, a patto che "puntatore" sia stato prima associato alla funzione moltiplica.
I puntatori a funzione possono rivelarsi strumenti duttili e utili, e permettono di creare codice sorprendentemente compatto, soprattutto se utilizzati in **array**.
Ad esempio, tenendo valide le definizioni precedenti delle funzioni, potremmo proporre una nuova versione della **calcolatrice minima** che ci ha spesso guidato nel libro "Imparare C++".

```
//calcolatrice minima con puntatori a funzione
#include <iostream>
using namespace std;

//[definizioni delle funzioni]

int main()
{
    double (*puntatore[]) (int, int) = {&somma,
&sottrai, &moltiplica, &dividi};
    char simbolo[] = {'+', '-', '*', '/'};

    cout << "Inserisci il primo numero:  ";
    int a; cin >> a;
    cout << "Inserisci il secondo numero: ";
    int b; cin >> b;

    for (int i=0; i<4; i++)
        cout << a << simbolo[i] << b << " =
" << (*puntatore[i])(a,b) << endl;
```

```
    return 0;
}
```

Questa versione è funzionante, e si rivela una scelta molto interessante per più motivi: innanzitutto abbiamo evitato il continuo riproponimento delle istruzioni cout, imbrigliando invece il tutto in un ciclo. Ciò permette una maggiore leggibilità e manutenibilità: se volessimo aggiungere una nuova operazione basterebbe definirla, aggiungerla all'array puntatore, e definirne un simbolo.
Anche queste operazioni, a ben vedere, potrebbero essere evitate: ad esempio, ti suggerisco come esercizio di provare a studiare un tipo di dati apposito (una struct, ad esempio), che riunisca in una sola voce il simbolo e il puntatore a funzione.
Il tutto sarà molto più coerente.

2.5.2 PUNTATORI A METODO

Con i normali puntatori a funzione non è possibile puntare a metodi di una classe, nemmeno se questi rispecchiano il prototipo del puntatore. La ragione è facilmente comprensibile, se si pensa alle differenze che intercorrono fra una funzione e un metodo – cosa della quale ci accorgeremo fra poco.
I puntatori a metodo seguono, invece, la sintassi dei puntatori a membro:

```
void (Cane::* puntatore) ()
```

Il codice qui sopra, ad esempio, dichiara un puntatore a un qualsiasi metodo della classe cane che restituisca void e non prenda in ingresso argomenti, come ad esempio **void faiVerso()**.
Al solito, il puntatore si referenzia come se il metodo (con indicazione della classe) fosse una variabile qualunque, ad esempio:

```
puntatore = &Cane::faiVerso;
```

Come nel caso dei puntatori a funzione, la chiamata del metodo può essere fatta dereferenziando il puntatore, fra parentesi.
E come nel caso dei puntatori a membro, questo non ha senso se non viene indicata l'istanza su cui il metodo si applica.
Ciò porta alla scrittura:

```
Cane c;
(c.*puntatore)(); //richiama il puntatore a metodo per c
```

Anche se comunemente vengono utilizzati molto poco, i puntatori a metodo possono portare nella programmazione a oggetti tutti i vantaggi che i puntatori a funzione offrono in quella di stampo procedurale.
Un esempio può essere la generazione di un Cane ammaestrato, capace di imparare dinamicamente delle sequenze di ordini, e di eseguirle in fila a comando.

```
class Cane {
public:
    //metodi
    void faiVerso() {cout << "Bau! ";}
    void dormi() {cout << "zzz... ";}
    void mordi() {cout << "sgnack! ";}

    //array di puntatori a metodo
    void (Cane::* ordine[3]) ();

    //metodo per eseguire gli ordini
    void eseguiOrdini() {
        for (int i=0; i<3; i++)
            (this->*ordine[i])();
    };
};
```

Qui la sequenza di ordini è ottenuta attraverso un **array di puntatori** a **metodo** (in questo caso il cane ha una "memoria" fissa di tre ordini), ed è previsto un metodo **eseguiOrdini** per effettuarla.
Possiamo provare ad istruire il nostro amico, in questo modo:

```
int main()
{
    Cane c;

    c.ordine[0] = &Cane::faiVerso;
    c.ordine[1] = &Cane::faiVerso;
    c.ordine[2] = &Cane::mordi;
    c.eseguiOrdini();
    return 0;
}
```

```
Bau! Bau! Sgnack!
```

Questa tecnica è molto interessante, perché permette ad un oggetto di memorizzare un informazione o una sequenza, come in questo caso, sui metodi (di se stesso o di un'altra classe) da richiamare in un secondo tempo, in maniera completamente dinamica. Basandosi su questo principio, alcuni programmatori traggono vantaggio dall'uso dei puntatori a metodo, usandoli su funzioni virtuali; ciò permette di implementare una sorta di polimorfismo alternativo: meno sicuro, ma più versatile.

2.6 RAPPRESENTAZIONE DEI METODI

Abbiamo visto la rappresentazione in memoria dei membri, ma che dire di quella dei metodi?
Se proviamo a prendere il nostro primo esempio rappresentato in fi-

gura 1.1, ci accorgiamo presto che la presenza dei metodi non prende alcuno spazio in memoria. Un oggetto di tipo Cane&, ad esempio, non ha membri, pertanto occupa un solo byte in memoria (gli oggetti devono prendere almeno un byte di spazio, dal momento che char è la minima unità di allocazione).
Riflettendoci un attimo, questo appare quasi ovvio: il fatto che un oggetto possieda dei metodi non implica che questo debba memorizzare chissà quali informazioni.
Ciò è evidente nel caso di funzioni dichiarate come static, le quali non hanno nemmeno accesso ai membri di un'istanza specifica. Mettendosi nell'ottica del compilatore una situazione come:

```
class A {
    static void metodo() {};
};
```

può essere benissimo riscritta esternamente alla classe, come:

```
class A {};
void metodo() {}; // A::metodo();
```

In questo caso, il compilatore dovrebbe solo fare attenzione alla visibilità del metodo, e al fatto che venga richiamato solo con una chiamata del tipo "A::metodo()".
Le funzioni comuni sono concettualmente molto simili ai metodi: l'unica differenza sta nel fatto che hanno effettivamente accesso ad un'istanza specifica di un oggetto. Il compilatore tipico risolve la situazione in modo semplice.
Ammettiamo di avere una situazione del genere:

```
class Classe {
public:
    int membro;
```

```
    void setMembro(int valore) {
        membro = valore;
    };
};

int main()
{
    Classe classe;
    classe.setMembro(5);
}
```

Il metodo setMembro agisce su un membro della Classe, pertanto è legato ad un'istanza ("classe"). Il compilatore può trasformare il codice così:

```
class Classe
{
public:
    int membro
}

void setMembro(const Classe* this, int valore)
//Classe::setMembro()
{
    this->membro = valore;
}

int main()
{
    Classe classe;
    Classe::setMembro(&classe, 5);
    return 0;
}
```

Il compilatore riesce così ad operare sui metodi della classe trattandoli come comunissime funzioni, in cui l'oggetto chiamante viene passato semplicemente come argomento "this".
Ogni riferimento interno alla funzione che non venga trovato in variabili locali o globali viene ricercato implicitamente nell'argomento this.
Come al solito, un minimo di attenzione alla visibilità è tutto ciò che è richiesto al compilatore (ovvero, rispondere alla domanda: "è pubblico o privato? si può richiamare questo metodo da qui?") per svolgere il passaggio in maniera coerente.

2.7 RAPPRESENTAZIONE DEI METODI VIRTUALI

2.7.1 EREDITARIETÀ IN CLASSI NON-POLIMORFE

Anche nel caso di classi derivate, la soluzione trovata precedentemente risulta comunque coerente.
Ogni volta che si richiama un metodo lo si fa da un oggetto preciso; questo ha un interfaccia che stabilisce in maniera chiara quale funzione usare.
Tutto può essere risolto a tempo di compilazione senza problemi anche in caso di ridefinizioni, come si vede dal seguente esempio:

```
int main() {
    Cane            fido;
    CaneDomestico   pluto;
    CaneRandagio    ringhio;
    cane.faiVerso();    //Cane::faiVerso(&fido);
    pluto.faiVerso();   //CaneDomestico::faiVerso(&pluto);
    ringhio.faiVerso(); //CaneRandagio::faiVerso(&ringhio);
}
```

Nemmeno un upcast mette in crisi il sistema: il compilatore richiama il metodo definito dalla classe base.

```
int main()
{
    CaneDomestico pluto;
    Cane& cane = static_cast<Cane&>(pluto);
    cane.faiVerso();    //Cane::faiVerso(&cane);
}
```

Non c'è nessun problema per il compilatore perché il metodo **faiVerso() non è definito come virtuale**, quindi il suo comportamento non è polimorfico.
Ma se Cane::faiVerso() fosse dichiarato come virtual, allora il comportamento riportato qui sopra non sarebbe accettabile.

2.7.2 EREDITARIETÀ IN CLASSI POLIMORFE

```
class Cane
{
public:
    virtual void faiVerso() {
        cout << "Bau!";
    };
};

int main()
{
    CaneDomestico pluto;
    Cane& cane = static_cast<Cane&>(pluto);
    cane.faiVerso();    //CaneDomestico::faiVerso(&cane);
}
```

In questo caso il compilatore deve richiamare il faiVerso() dell'interfaccia CaneDomestico, e per far ciò deve per forza aver prima inserito da qualche parte in memoria un informazione sul tipo originale della variabile cane (CaneDomestico, appunto).
Questo punto è fondamentale: classi senza metodi virtuali non hanno questo problema. Le classi con almeno un metodo virtuale devono definire un modo per "ricordare" la loro reale appartenenza e vengono dette **polimorfe**.

2.8 VPTR E VTABLE

2.8.1 VPTR

Un primo indizio fondamentale nella "caccia all'informazione nascosta dal compilatore" è fornito dall'operatore sizeof: la versione virtuale del Cane prende ben 4 bytes, contro l'unico byte (peraltro "pro forma") precedente. Evidentemente **alle classi dinamiche viene aggiunto almeno un membro**.

CaneDomestico e CaneRandagio hanno aumentato la loro dimensione di 4 unità; poiché ereditano da una classe virtuale, sono a loro volta classi virtuali.
Ad un'analisi col debugger di Visual C++ (vedi figura 2.1) si scopre che il "membro nascosto" di Cane è segnalato con la sigla **vfptr**, che sta per **Virtual Function Pointer**, e che in molti (me compreso) chiamano "semplicemente" **vptr**.

2.8.2 VTABLE

Ogni vptr punta ad una diversa vftable, sigla che sta per **Virtual Function Table**, e che in molti (me compreso) chiamano "semplicemente" **vtable**.
Questa contiene un elenco di indirizzi alle funzioni virtuali definite dalla classe.
Se Cane fosse definito così, ad esempio:

```
class Cane
{
public:
    virtual void faiVerso() {cout << "Bau!";}
    virtual void dormi()      {cout << "zzz...";}
    virtual void mordi()    {cout << "sgnack!";}
    void annusa() {cout << "sniff";}
}
```
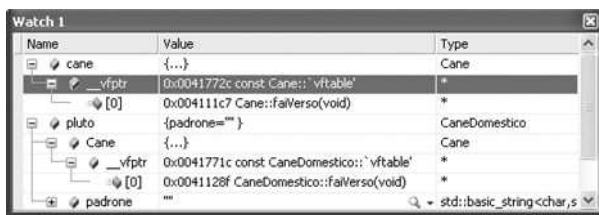
Figura 2.1: Vptr e Vtable visti da un debugger

la vtable di Cane conterrebbe gli indirizzi delle tre funzioni virtuali (respira è "statica" e non ha bisogno di rientrare nella tabella).
Sempre tenendo buona questa definizione di Cane, proviamo a dare una definizione di CaneDomestico:

```
class CaneDomestico : public Cane
{
public:
    string padrone;

    void faiVerso() {cout << "Arf!";}
    void dormi()   {cout << "ronf...";}
};
```

CaneDomestico ridefinisce due delle tre funzioni virtuali.

La sua vtable conterrà comunque tre puntatori: "mordi()" farà riferimento alla stessa funzione esposta dall'interfaccia di Cane, come mostrato dalla figura 2.2.
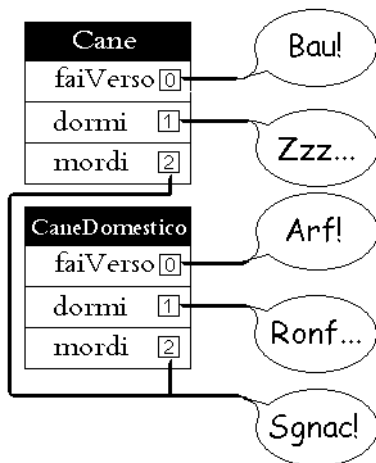
Figura 2.2: Rappresentazione delle vtable di Cane e CaneDomestico

I vari puntatori a funzione delle vtable possono, quindi, essere visti ancora una volta come degli spiazzamenti (i numeri indicati in figura 2.2 a fianco dei nomi).

2.8.3 LATE BINDING

Vediamo nel dettaglio come il compilatore usa vptr e vtable per implementare correttamente il comportamento delle classi polimorfe. Seguiamo questo codice passo passo:

```
int main()
{
    CaneDomestico pluto;
```

```
    CaneRandagio ringhio;
    Cane* cane[2] = {&pluto, &ringhio};
    cane[0]->faiVerso();
    cane[1]->faiVerso();
    return 0;
}
```

Nella prima riga viene dichiarato pluto come CaneDomestico.
Dal momento che questa classe eredita da Cane (la quale è una classe polimorfa), alla variabile viene assegnato un vptr specifico, che punta alla vtable di CaneDomestico.
La stessa informazione è contenuta per ringhio, collegata alla vtable di CaneRandagio.
Nella terza riga viene definito un array di puntatori di classe Cane, che viene inizializzato.
Nell'inizializzazione avvengono degli upcast impliciti, equivalenti a:

```
cane[0] = static_cast<Cane*>(&pluto);        //vtable di CaneDomestico
cane[1] = static_cast<Cane*>(&ringhio);//vtable di CaneRandagio
```

Il compilatore trasforma le due righe successive in questo modo:

```
prendi l'elemento 0 di "cane" e richiama la funzione 0 (faiVerso) della
vTable memorizzata nel suo vptr.
prendi l'elemento 1 di "cane" e richiama la funzione 0 (faiVerso) della
vTable memorizzata nel suo vptr.
```

Questo mostra in maniera evidente perché le chiamate a variabili puntatore di classi polimorfe siano più lente: l'esecuzione di tutti questi passaggi aggiuntivi richiede più tempo della semplice chiamata a funzione.
Il meccanismo per cui i riferimenti sono risolti a tempo di esecuzione si chiama **late binding**, o **binding dinamico**.

Per contrapposizione, quello per cui tutti i riferimenti sono risolti a tempo di compliazione si chiama **early binding**, o **binding statico**.

2.9 DYNAMIC_CAST

Ti ricordi del problema che aveva l'Accalappiacani che abbiamo conosciuto nel paragrafo 1.5.1?
A questo punto Tizio potrebbe avere un'idea maliziosa: "se il compilatore dietro le scene registra delle informazioni sulle classi polimorfe, posso controllare di nascosto se il downcast si può fare o no!".
Questo è il suo piano:

```
Rendo Cane una classe polimorfa, così il compilatore dovrà associarvi le
informazioni dinamiche.
Prendo "cane" (l'istanza che mi viene passata) e controllo il suo vptr.
Se questo punta ad una vtable compatibile con CaneDomestico, eseguo il
downcast.
Altrimenti, evito di riportare il cane al padrone.
```

L'idea non è affatto male, ma Tizio avrebbe il suo daffare a rintracciare l'esatto indirizzo del vptr, e anche se ci riuscisse il suo piano non sarebbe compatibile con altri compilatori, i quali dovranno sì nascondere la stessa informazione, ma probabilmente lo faranno in altro luogo, o in altri modi.
Per questo, Il C++ ufficializza quest'idea nell'operatore di conversione **dynamic_cast**, che si usa comunemente proprio nel caso in cui si voglia forzare un downcast da una classe base polimorfa, controllando se la classe di destinazione è compatibile.
Se hai esprerienza del modello COM, può esserti utile vedere la dinamica di questo controllo come un'operazione di "query interface". La sintassi di dynamic_cast è:

```
dynamic_cast<ClasseDiDestinazione*>
```

```
(IstanzaDiClasseBasePolimorfa*)
```

Poiché dynamic_cast si usa per ottenere un comportamento polimorfo, il tipo di destinazione dev'essere per forza un puntatore (come nella sintassi precedente), o un riferimento (come nella sintassi seguente).

```
dynamic_cast<ClasseDiDestinazione&>
(IstanzaDiClasseBasePolimorfa*)
```

In entrambi i casi l'effetto è quello di forzare dinamicamente un downcast: se quest'operazione è possibile, dynamic_cast restituisce una variabile appartenente alla classe di destinazione, altrimenti restituisce 0.
Nel caso in cui si usi un riferimento, invece, un'operazione non valida restituirà un'eccezione di tipo **bad_cast** (vedi capitolo 3). Ecco una possibile soluzione al problema dell'Accalappiacani, grazie a dynamic_cast:

```
//Cane viene definita come classe polimorfa
class Cane {
public:
    virtual void faiVerso() {cout << "Bau!";}
};

// […] definizioni di CaneDomestico, CaneRandagio, Accalappiacani

void Accalappiacani::recupera(Cane& cane)
{
    //downcast dinamico
    CaneDomestico* dCane = dynamic_cast<CaneDomestico*>(&cane);

if (dCane) //se la conversione è avvenuta con successo
```

```
            cout<< "vado a riportare il cane a "
                  << dCane->padrone << endl;
else   //altrimenti, non è un cane domestico (non ha padrone)
            cout<< "vado a portare il cane al canile" << endl;
}
int main()
{
    CaneRandagio       ringhio;
    CaneDomestico      pluto;
    pluto.padrone = "Topolino";
    Accalappiacani tizio;
    tizio.recupera(ringhio);
    tizio.recupera(pluto);
    return 0;
}
```

```
vado a portare il cane al canile
vado a riportare il cane a Topolino
```

Il fulcro del codice è il downcast dinamico da Cane* a CaneDomestico*. All'accalappiacani basta controllare se questa conversione può avvenire: in tal caso può accedere al membro padrone. In caso contrario, la variabile dCane sarà pari a 0, e quindi il programma eseguirà l'istruzione in else. Vale ancora una volta la pena di ribadire che **dynamic_cast può essere usato soltanto con classi polimorfe**: classi senza metodi virtuali, infatti, a tempo di esecuzione non memorizzano alcuna informazione sul tipo.

2.10 RTTI

Il C++ permette di ottenere delle informazioni sulle classi che vengono utilizzate: sizeof ne è un esempio, ma come abbiamo visto opera in maniera statica, a tempo di compilazione. Grazie agli accorgi-

menti visti in questi capitoli, in C++ è anche possibile ottenere informazioni aggiuntive a tempo di esecuzione: queste vengono chiamate **RTTI** (Real Time Type Informations, cioè informazioni sul tipo in tempo reale), sono una caratteristica comparsa molto di recente nello standard, e prevedono l'uso dell'operatore **typeid** e della classe **type_info**.

2.10.1 TYPEID SU TIPI NON-POLIMORFI

Potrei dirti che typeid è un operatore speciale, che può essere utilizzato soltanto per confrontare fra loro due oggetti e vedere se appartengono allo stesso tipo; anche se questa definizione è po' forzata e semplicistica, resta comunque un ottimo approccio per introdurre la questione.
Analogamente a sizeof, typeid può essere richiamato per un tipo o per un espressione, cosicché si può scrivere:

```
Frazione f;
typeid(f) == typeid(Frazione); //vero
```

In questo caso, viene eseguito un confronto fra un tipo e il tipo di un oggetto. In altre parole, stiamo chiedendo: f è una Frazione?
Poiché la risposta è sì, il confronto restituisce vero.
Possiamo usare typeid anche per altri casi analoghi:

```
Frazione f, g;
typeid(Frazione) == typeid(Cane) // tipo == tipo (falso)
typeid(f) == typeid(Frazione)       // espressione == tipo (vero)
typeid(f) == typeid(g)       // espressione == espressione (vero)
typeid(f+g) == typeid(g)   // espressione == espressione (vero)
typeid(int) == typeid(long)          // espressione == espressione (falso)
```

L'ultimo confronto è interessante, perché mostra che due tipi sono diversi indipendentemente dal fatto che esista una conversione impli-

cita (per promozione, costruttore, operatore di conversione definito…). Un altro confronto interessante è questo:

```
Frazione f;
Frazione* ptrf = &f;
typeid(f) == typeid(ptrf)    //espressione == &espressione (falso)
typeid(Frazione) == typeid (Frazione*) // tipo == tipo* (falso)
```

I due confronti sono sostanzialmente identici, perché mettono in relazione un tipo (nell'esempio, Frazione) con un suo puntatore (nell'esempio, Frazione*): ciò restituisce sempre falso.
Bada che questo è vero per i puntatori, ma non per le reference, dal momento che un riferimento è praticamente un alias.
Pertanto il seguente confronto restituisce vero.

```
Frazione f;
Frazione &reff = f;
typeid(f) == typeid(reff)    //espressione == riferimento (vero)
typeid(Frazione) == typeid(Frazione&) // tipo == tipo& (vero)
```

Tutte questi confronti sono semplici sia per l'uomo che per il compilatore, che ha vita facile nel rintracciare le informazioni in maniera statica.
Ma quando si ha a che fare con puntatori e downcasting, le cose vanno interpretate con un po' di attenzione.
Ad esempio, se un puntatore di classe base non polimorfa punta ad un oggetto di classe derivata, il tipo del puntatore e quello dell'oggetto non saranno uguali. Esempio:

```
//In questo esempio, Cane non è una classe polimorfa.
class Cane {
    void faiVerso() {cout << "Bau!";}              //non è virtuale
}
```

```
CaneDomestico pluto;
Cane& cane = pluto;
typeid(cane) == typeid(pluto)    //(Cane& == CaneDomestico)? falso!
```

Nell'esempio, il metodo faiVerso() non è definito come virtuale, pertanto Cane è una classe non polimorfa.
L'oggetto cane punta effettivamente a un CaneDomestico, ma a typeid questo non interessa. Perché ne tenga conto, è necessario che Cane sia un tipo polimorfo.

2.10.2 TYPEID SU TIPI POLIMORFI

Per tipi polimorfi, il compilatore usa typeid con l'accortezza di andare a cercare, grazie ai vptr e alle vtable, le informazioni sul tipo realmente puntato da un riferimento o un puntatore.
Per quanto riguarda i riferimenti ciò è facilmente dimostrabile con la versione polimorfa dell'esempio precedente:

```
//In questo esempio e nei successivi, Cane è una classe polimorfa.
class Cane {
    virtual void faiVerso() {cout << "Bau!";}   //è virtuale
}

CaneDomestico pluto;
Cane& cane = pluto;
typeid(cane) == typeid(pluto)    //(Cane& == CaneDomestico)? vero!
```

La cosa è analoga per quanto riguarda i puntatori: bisogna solo ricordarsi di **dereferenziarli**, per ottenere l'oggetto corretto. In caso contrario, infatti, si otterrà un semplice puntatore al tipo base:

```
CaneDomestico pluto;
Cane* cane = &pluto;
typeid(cane) == typeid(pluto)    //(Cane* == CaneDomestico)? falso!
```

```
typeid(*cane) == typeid(pluto)   //(CaneDomestico == CaneDomestico)?
vero!
```

2.10.3 BAD_TYPEID

Occorre fare attenzione, quando si gioca con i puntatori, che questi siano validi: se si è dichiarato un tipo polimorfico, lo si è referenziato con un puntatore e questo è messo a terra, typeid() rischia di far accedere ad un vptr inesistente.
Per questo, lo standard definisce che un'operazione simile debba generare un'eccezione di tipo **bad_typeid** (vedi capitolo 3).

```
Cane* cane = 0;
typeid(*cane)      //bad_typeid
```

2.10.4 TYPE_INFO

A questo punto hai capito la dinamica dei confronti fra tipi, ma probabilmente avrai ancora un mucchio di dubbi irrisolti su questo fantomatico operatore typeid. Posso immaginarne qualcuno: "Da dove salta fuori? Che cosa restituisce esattamente?
Perché hai detto che serve solo per fare confronti? Perché hai detto che non è del tutto vero?". Bravo, belle domande!
Vedrò di rispondere con ordine.
Innanzitutto, typeid è parte dello standard del linguaggio C++, anche se non da molto.
Questo significa che puoi usarlo come un operatore qualsiasi, senza bisogno di includere alcuna libreria, e adoperarlo per fare confronti fra tipi.
Ma per poter avere accesso diretto a ciò che restituisce, devi includere l'header <typeinfo>, che contiene alcune dichiarazioni appartenenti al namespace std.
La più importante è la classe **type_info**, che è proprio il tipo restituito da typeid. La definizione di questa classe è a discrezione del compilatore, però solitamente segue questa struttura:

```
class type_info {
private:
    //costruttori privati
    type_info& operator=(const type_info&);
    type_info(const type_info&);

//nome della classe
const char *_name;
public:
//restituisce il nome della classe
const char* name() const
    { return _name; }

    //operatori di confronto
    bool operator==(const type_info&  b) const;
    bool operator!=(const type_info& b) const
    {return !operator==(b);}

    //definizione della classe bad_typeid…
    //definizione di altro a discrezione del compilatore (before, etc…)
};
```

Come ci aspettavamo, sono presenti gli operatori di uguaglianza e di disuguaglianza. Il costruttore per copia e l'operatore d'assegnamento privati sono una finezza di design, che permette di creare classi che non sono direttamente istanziabili, né duplicabili; l'unico modo di ottenere un type_info è per mezzo dell'operatore typeid, pertanto ci si può accontentare, al massimo, di associarne il risultato a un riferimento o un puntatore – del resto, alterare il contenuto di un type_info sarebbe un atto di vandalismo insensato.
L'unico attributo standard di type_info è il nome della classe, richiamabile attraverso la funzione name(), che restituisce una stringa in stile C.

Questo può essere utile se vuoi creare un framework capace di mostrare il nome delle classi, o più semplicemente se vuoi sapere velocemente il tipo di un espressione, il che spesso non è così immediato. Ad esempio, sapresti dire che cosa stampa a video questo codice?

```
#include <typeinfo>
#include <iostream>
class A {public: virtual ~A() {};};
class B : public A {};
int main()
{
    B b;
    A* ptr_b = &b;
    std::cout << typeid(ptr_b).name();
    return 0;
}
```

2.11 CONCLUSIONI

In questo capitolo abbiamo visto alcune delle possibilità più avanzate offerte dal C++, e quali artifici si celino effettivamente dietro le scene.
Ciò è fondamentale per avere una comprensione avanzata e attiva del linguaggio: meno si crede alla “magia” in certe faccende, e meglio è.
Devo anche dirti che molti di questi argomenti sono controversi e poco conosciuti ai programmatori meno esperti, e che potrai trovare persino compilatori che non si sono ancora aggiornati per implementarli (una buona discriminante per valutarne la qualità, per inciso).
Scoprirai anche che è possibile programmare intere suite di applicazioni senza far ricorso a molte delle caratteristiche presentate, e questo è un bene: conoscere vicoli oscuri e alternativi può spesso rive-

larsi indispensabile all'occorrenza, ma ciò non toglie che bisognerebbe percorrere, quando possibile, le strade illuminate, comode e sicure. Ad esempio, a cosa può portare il modello "dell'Accalappiacani" che abbiamo visto in questo capitolo? All'espandersi della complessità del mondo rappresentato, è possibile ipotizzare qualcosa del genere:

```
void Accalappiacani::Recupera(const Cane& cane)
{
    if (typeid(cane) == typeid(CaneConMedaglietta)) {codice};
    else if (typeid(cane) == typeid(CaneConChip)) {codice};
    else if (typeid(cane) == typeid(CaneConMarchio)) {codice};
    else if (typeid(cane) == typeid(CanePericoloso)) {codice};
    //eccetera…
}
```

Questo modello è fallimentare perché introduce tutta la complessità del mondo degli oggetti senza trarne alcun vantaggio effettivo: né semplificazione del codice, né disaccoppiamento, né incapsulamento. Questo tipo di codice andrebbe forse bene in C, o in un linguaggio poco espressivo in cui è necessario lasciarsi andare a switch chilometrici, ma programmare OOP è un'altra cosa. Di fronte a simili situazioni, il primo pensiero da cogliere è che il design è concettualmente sbagliato. Ecco una possibile soluzione più "illuminata":

```
class Cane
{
Identificatore* id;
virtual void faiVerso();
}
class Identificatore
{
    Cane& cane;
```

```
    virtual cane_info getInfo() = 0;
}
class cane_info
{
    string proprietario;
    bool pericoloso;
    //etc…
}
```

Con questo sistema un cane porta addosso un Identificatore: una classe virtuale pura che dev'essere ereditata dai vari sistemi (Medaglietta, Marchio, etc…): il vincolo di associazione fa si che un cane possa non avere alcun identificatore, semplicemente ponendo il puntatore a 0.
Se l'identificatore c'è, all'Accalappiacani basta richiamarne il metodo getInfo per avere tutte le informazioni che desidera:

```
void Accalappiacani::recupera(Cane& cane)
{
    if (cane.id)
        cout << "riporto il cane a " << cane.id.getInfo().padrone;
    else
        cout << "porto il cane al canile";
}
```

Questo è solo uno degli innumerevoli design possibili: saper scegliere il migliore fa parte dell'esperienza e della preparazione individuale. Ma è pur vero che il mondo reale è spesso fatto di vicoli bui: codice imperfetto, situazioni impreviste, compiti particolari, librerie immutabili e vincoli prestazionali. In questi casi saper trovare soluzioni estemporanee, sporcandosi le mani con gli elementi più avanzati del linguaggio fa la vera differenza fra il "semplice programmatore" e il "l'esperto di C++".

ECCEZIONI

Questo capitolo parlerà delle eccezioni, cioè di quegli eventi imprevisti che si possono verificare durante l'esecuzione del codice, e che altri (le librerie, l'ambiente, il computer) non riescono a gestire per noi. Solitamente l'effetto più comune al quale si pensa parlando di eccezioni è il crash dell'applicazione. Ho spesso ripetuto che c'è di peggio: la trasmissione silenziosa dell'errore.
Le funzioni che cercano di tirare avanti comunque, ben sapendo che qualcosa non va (ne ho dato vari esempi nel corso di "Imparare C++") trasmettono in maniera silenziosa una situazione scorretta, nella speranza che qualcuno se ne accorga e vi ponga rimedio, o la aggiri alterando il flusso dell'esecuzione.
Questa filosofia dell'errore porta spesso ad un 'effetto valanga' di piccole anomalie che culminano infine in un comportamento scorretto dell'applicazione – e ci sono campi in cui gli effetti di un anomalia si pagano realmente cari. Il classico esempio di questo stile è la logica di trasmissione del valore NULL nei database, che viene ferocemente osteggiata da molti programmatori. Il C++, invece, ha un approccio diametralmente opposto: se c'è un errore e nessuno lo gestisce, l'applicazione va in crash. Punto.
Prevedere la gestione degli errori in un'applicazione medio-grande diventa quindi una necessità, se si vuole scrivere un programma robusto che tenga conto della realtà e degli imprevisti.

3.1 LA CLASSE VETTORE

Nel libro "Imparare C++", nel quale abbiamo visto un esempio di un'ipotetica classe Vettore, in grado di gestire in maniera dinamica oggetti di tipo intero. Poiché la stessa classe ci farà da sostegno per questo capitolo e per il prossimo, e dal momento che potresti non possedere il libro in questione, riporto qui il suo codice sorgente:

```
#include <iostream>
```

```
class Vettore
{
private:
    int* elementi;
    int grandezza;

public:
    Vettore(int g=1) : grandezza(g)
    {
        //crea i nuovi elementi
        elementi = new int[grandezza];
        //inizializza ogni elemento a zero
        for (int i=0; i<grandezza; i++)
            elementi[i] = 0;
    }
    //costruttore per copia [6.5]
    Vettore(const Vettore& b) : grandezza(b.grandezza)
    {
        //crea i nuovi elementi
        elementi = new int[grandezza];
//copia ogni elemento
        for (int i=0; i<grandezza; i++)
            elementi[i] = b.elementi[i];
    }

    ~Vettore()
    {
        if (elementi) { //se il puntatore è valido
            delete[] elementi;  //elimina
            grandezza = 0;      //neutralizza
            elementi  = 0;      //neutralizza
        }
    }
```

```
    //overloading dell'operatore di assegnamento [6.6.5]
    Vettore operator=(const Vettore& b)
    {
        //distrugge i vecchi elementi
        grandezza = b.grandezza;
         if (elementi)
            delete[] elementi;
        //copia i nuovi elementi
        elementi = new int[grandezza];
        for (int i=0; i<grandezza; i++)
            elementi[i] = b.elementi[i];
        return *this; //costruttore per copia
    }
    //Overloading dell'operatore []
    int& operator[](int pos)
    {
        if (pos < grandezza && pos >= 0)
            return elementi[pos];
        else {
            std::cout << "Errore: indice fuori dai margini";
            return elementi[0];
        }
    }
    int getGrandezza() {return grandezza;}
};
```

3.2 SOLLEVARE UN'ECCEZIONE

Per inquadrare il problema delle eccezioni in maniera corretta, bisogna scindere il proprio sguardo in due prospettive complementari: la prima è quella di chi sta creando una libreria, o una classe, e si trova in una condizione in cui sa che qualcosa sta andando storto, ma non sa come rimediare.

La seconda è quella di chi si affida ad una libreria sapendo come gestire eventualmente l'errore, ma non sapendo con precisione quale errore potrà incontrare.
È proprio in virtù di questa suddivisione dei compiti che la classe Vettore si presta tanto bene a far da esempio sull'uso delle eccezioni: è sufficientemente astratta e riutilizzabile da poter essere considerata parte di un'ipotetica libreria.
Per cominciare, quindi, proviamo a metterci nei panni di chi scrive la libreria, e poniamoci la domanda fatidica: "esiste una qualche richiesta dall'utente che, combinata ad una certa situazione, possa eventualmente causare un errore?".
La risposta è quasi sempre sì, e gli esempi aumentano in modo proporzionale al coefficiente di paranoia con cui si analizza la questione.
Nel nostro caso, abbiamo già individuato un errore netto, che si ha quando un utente si trova a richiedere un elemento con indice superiore alla grandezza del Vettore:

```
int& operator[](int pos)
{
    if (pos < grandezza && pos >= 0)
        return elementi[pos];
    else
{
        std::cout << "Errore: indice fuori dai margini";
        return elementi[0];
    }
}
```

In altre parole, un classico degli array: indice fuori dai margini. Per come è progettata adesso, la nostra libreria non si comporta proprio bene: riesce nell'arduo compito di portare a termine tre azioni sbagliate in sole due righe:

- **Blatera inutilmente** sullo schermo (verso l'utente dell'applicazione, che non è minimamente interessato alla cosa), che c'è stato un errore. Tale informazione, peraltro, è del tutto inutile anche per il debug, perché è priva di un contesto: c'è stato un errore? Dove? Quando? Perché?
- In caso di errore, restituisce l'elemento zero, **fornendo così un valore scorretto al chiamante**.
- **Non permette di sapere se l'operazione è andata a buon fine**, riducendo così all'impotenza anche il programmatore armato delle migliori intenzioni.

Questo non va bene, perché in questo turno del Grande Gioco delle Eccezioni è lo scrittore della libreria ad avere informazioni sull'errore in questione, e a doverlo notificare all'utente.
Ma come? Vediamo come possiamo usare il meccanismo delle eccezioni per migliorare la situazione: innanzitutto definiamo una classe per riportare l'errore.

```
class IndiceFuoriDaiMargini {};
```

a questo punto possiamo **sollevare** l'eccezione, utilizzando la parola chiave **throw**, con un'istanza della nostra eccezione come argomento:

```
int& operator[](int pos) {
    if (pos < grandezza && pos >= 0)
        return elementi[pos];
    else
        throw(IndiceFuoriDaiMargini());
}
```

Abbiamo già eliminato due difetti: la libreria non blatera più, e non viene restituito un elemento scorretto. Invece, **throw** allerta la fun-

zione chiamante con un errore di tipo **IndiceFuoriDaiMargini**: se questa ha previsto (vedremo presto **come**) un gestore apposito, il controllo passerà a lei. Altrimenti, si andrà a vedere se il "chiamante del chiamante" ha previsto una gestione degli errori. E così via, ricorsivamente, fino ad srotolare completamente lo stack delle chiamate. Se si ritorna a **main** senza che sia previsto un gestore degli errori per quella specifica eccezione, verrà richiamata la famigerata funzione **std::terminate()** o dei similari più verbosi stabiliti dal compilatore, i cui effetti puoi osservare in figura 3.1.

```
his application has requested the Runtime to terminate it in an unusual wa
lease contact the application's support team for more information.

ress ENTER to continue.
```
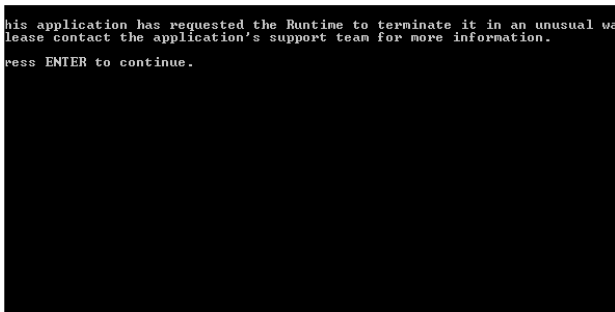
Figura 3.1: Un'eccezione non è stata gestita

3.3 GESTIRE UN'ECCEZIONE

Ora proviamo a metterci dall'altra parte della barricata.
Siamo dei programmatori fiduciosi e un po' sprovveduti che vogliono usare la classe Vettore. Scriviamo questo programma di test:

```
#include <iostream>

int main()
{
    Vettore v(10);          //10 elementi in tutto
    int n = v[20]; //ventunesimo elemento?
```

```
        std::cout << n; //inutile: non sarà mai eseguito

    return 0;
}
```

E il risultato dell'esecuzione sarà immancabilmente quello presentato in figura 3.1. È colpa nostra! Avremmo dovuto sapere che un'operazione del genere può causare delle eccezioni, anche se a nostra scusante c'è il fatto che l'odioso creatore della libreria (sempre noi, peraltro) non ci ha fatto pervenire in nessun modo quest'informazione (vedi paragrafo 3.7). Dunque, studiamo la libreria, e vediamo che **Vettore** può sollevare un'eccezione di tipo **IndiceFuoriDaiMargini**; pertanto proviamo a gestirla tramite un **blocco try{} catch{}**. Il costrutto ha la seguente sintassi:

```
try {
    istruzioni pericolose
}
catch(tipoEccezione)
{
    istruzioni da eseguire se l'eccezione si verifica
}
```

Alla prima eccezione verificatasi nel blocco try, verrà immediatamente richiamato il blocco catch, se questo prevede la gestione di un errore compatibile.

```
#include <iostream>

int main()
{
    try {
        Vettore v(10); //10 elementi
```

```
            v[20] = 10;
            std::cout << "Accesso Riuscito"; //non sarà mai eseguita
      }
      catch(IndiceFuoriDaiMargini)
      {
            std::cout << "Wow: la gestione degli errori funziona!";
      }

      return 0;
}
```

Il confortante responso dell'esecuzione di questo programma è:

```
Wow: la gestione degli errori funziona!
```

Nelle implementazioni reali il blocco catch viene sostituito da un codice capace di gestire la situazione d'errore.
Nota, peraltro, che il sollevamento dell'eccezione provoca immediatamente il salto al blocco catch.
Pertanto le righe successive a quella che ha provocato l'errore non saranno mai eseguite.

3.4 GESTORI MULTIPLI

In molti casi una chiamata pericolosa può generare molte eccezioni di tipo diverso: basti pensare all'accesso ad un file.
Questo può essere già aperto o inesistente, o il disco può essere pieno o essere stato rimosso durante la scrittura, e così via.
Ogni eccezione di tipo diverso viene fatta ricadere sotto una classe diversa.
Pertanto, sarà necessario implementare più blocchi catch.
Poniamo, ad esempio, di passare un file ad un interprete, che prevede le seguenti eccezioni:

FileNonTrovato, GrammaticaNonTrovata, ErroreLessicale, ErroreDiSintassi.

```
int main()
{
    Interprete interprete;
    //inizializza interprete...

    try {
        interprete.analizza();
        interprete.esegui();
    }
    catch(FileNonTrovato) {
        //gestisce l'errore
    }
    catch(GrammaticaNonTrovata) {
        //gestisce l'errore
    }
    catch(ErroreLessicale) {
        //gestisce l'errore
    }
    catch(ErroreDiSintassi)
{
        //gestisce l'errore
    }

    return 0;
}
```

Se viene generata un'eccezione, il programma comincerà a cercare una corrispondenza dalla prima all'ultima: vale a dire che se l'errore è di tipo **FileNonTrovato**, gli altri catch non saranno neanche presi in considerazione.

3.5 GESTORE GENERICO

Talvolta non siamo interessati a conoscere esattamente quale eccezione si è verificata: basta sapere che qualcosa non è andato per il verso giusto, per annullare l'operazione.
In questi casi è possibile utilizzare i tre puntini di sospensione come argomento di catch.

```
int main()
{
    Interprete interprete;
    //inizializza interprete...

    try
{
        interprete.analizza();
        interprete.esegui();
    }
    catch(FileNonTrovato) {
        //gestisce l'errore FileNonTrovato
    }
    catch(...)
{
        //gestisce tutti gli altri errori
    }
    catch(ErroreLessicale) {
        //questo codice non sarà mai richiamato!
    }
    catch(ErroreDiSintassi) {
        //questo codice non sarà mai richiamato!
    }

    return 0;
}
```

Nell'esempio riportato, ho mostrato tutti i casi possibili: le eccezioni vengono valutate partendo dalla prima (**FileNonTrovato**).
Quando si arriva ai puntini di sospensione, l'entrata nel blocco catch viene forzata.
Così facendo, gli eventuali gestori successivi a quello generico non saranno mai richiamati - la maggior parte dei compilatori non fa fatica ad avvertire dello sbaglio.

3.6 GERARCHIE DI CLASSI ECCEZIONE

Una delle possibilità più interessanti offerte dalle eccezioni è quella di adoperare l'ereditarietà per creare delle vere e proprie gerarchie di eccezioni.
La cosa risulta molto comoda e vantaggiosa, perché in questo modo è possibile controllare un'intera famiglia di eccezioni attraverso un solo gestore.
Ad esempio, le eccezioni descritte in 3.4 possono essere poste secondo la gerarchia mostrata in figura 3.2.
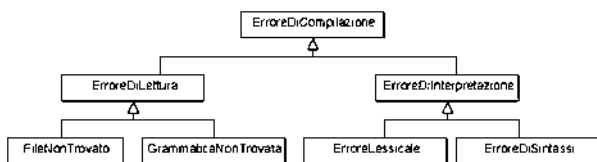
Figura 3.2: La famiglia di eccezioni ErroreDiCompilazione

Un codice del genere è perfettamente valido:

```
int main()
{
    Interprete interprete;
```

```
    try {
        interprete.analizza();
    }
    catch(ErroreDiLettura) {
        //chiede all'utente di specificare nuovamente
        //i file di riferimento
    }
    catch(ErroreDiInterpretazione) {
        //Notifica all'utente che è avvenuto un errore
        //di interpretazione
    }
}
```

In questo caso, potrebbe essere interessante definire meglio le classi di tipo ErroreDiInterpretazione, perché riescano a riportare anche dei dati circa la riga in cui si è verificato l'errore.

3.7 SPECIFICA DELLE ECCEZIONI

Qualche paragrafo addietro abbiamo definito "odioso" l'autore della libreria Vettore, perché non fa sapere in alcun modo ai suoi utenti quali errori è lecito aspettarsi dalla chiamata delle funzioni delle sue classi. La cosa è irritante perché se nella dichiarazione di una funzione non viene specificato nulla, come in:

```
void funzioneTipica();
```

questa ha il potenziale di generare qualsiasi tipo di eccezione, il che obbliga il chiamante a considerare casi che probabilmente non sono minimamente previsti, o a spulciare chilometri di documentazione alla ricerca di informazioni aggiuntive. In realtà il programmatore che scrive una libreria o una classe è sempre cosciente dei punti più critici del suo programma: se non fa uso dei contenitori, è diffi-

cile che venga generata un'eccezione di tipo **bad_range**!
La gestione delle eccezioni, insomma, è efficace se si sa **che cosa** gestire.
A tal fine il C++ permette di definire delle **specifiche delle eccezioni**, ovverosia di dichiarare di seguito all'intestazione della funzione o del metodo, l'elenco delle eccezioni che la classe può sollevare. Ad esempio:

```
void funzioneTipica() : throw(ErroreTipico, ErroreClassico);
```

Questo prototipo indica che la funzioneTipica() può sollevare solo due tipi di eccezioni: l'ErroreTipico e l'ErroreClassico. Pertanto sapremo che è una situazione che prevede cautela, e faremo tesoro dell'informazione per poter organizzare dei gestori opportuni. Questo porta logicamente ad una considerazione interessante: funzioni che si presume siano sicure (o, per dirla meglio, **non sollevino eccezioni**), possono essere definite con un throw privo di argomenti.

```
void funzioneSicura() : throw();
```

3.8 GESTORI CHE RILANCIANO ECCEZIONI

Talvolta non siamo sicuri di riuscire a domare l'eccezione con un gestore, o possiamo riuscirci solo parzialmente.
In questi casi, quando ci accorgiamo che non è possibile gestire la situazione, possiamo risollevare l'eccezione e delegare il controllo a qualche altro blocco catch. Il C++ prevede a questo fine l'utilizzo della parola chiave throw (priva di parentesi e argomenti).
Quando viene usata in questo modo, throw implica che l'ultima eccezione lanciata verrà sollevata nuovamente.Esempio:

```
int main()
```

```
{
    try {
        azionePericolosa();
    }
    catch(ClasseEccezione& e)
    {
        if (!sonoCapaceDiGestire(e))
            throw; //l'eccezione viene rilanciata
    }
    //... altri gestori che si occuperanno di gestire il rilancio…
}
```

3.9 ECCEZIONI PREDEFINITE

Il C++ prevede quattro classi di eccezioni:

- **bad_alloc**: richiamata quando un'operazione di allocazione dinamica (new) ha risultati imprevisti.
- **bad_cast**: richiamata quando un'operazione di dynamic_cast ha risultati imprevisti (vedi paragrafo 2.9).
- **bad_typeid**: richiamata quando un'operazione di risoluzione del tipo ha risultati imprevisti (vedi paragrafo 2.10).
- **bad_exception**: richiamata quando viene sollevata un'eccezione non prevista dalle specifiche (vedi paragrafo 3.7).

Ognuna di queste deriva direttamente da una classe base chiamata **exception**, che è anche la base delle eccezioni della libreria standard, ed è (semplificando un po') definita così:

```
class exception
{
public:
    exception() throw() { }
```

```
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

Se sei un novizio, forse avrai dei problemi a interpretare l'ultima riga: "la funzione **what**() è ridefinibile (**virtual**), restituisce una stringa costante (**const char***), è un metodo costante (**const**) e non lancia eccezioni (**throw**())". In effetti, se l'eccezione stessa si mettesse a sollevare eccezioni sarebbe grave. È possibile far derivare le **proprie** eccezioni da std::exception, sfruttandone così l'interfaccia.

```
#include <iostream>
#include <string>

using namespace std;

//classe che gestisce un errore lessicale
class ErroreLessicale : public exception
{
string parola;
public:
    ErroreLessicale(string p) throw() : parola(p) {}
    ~ErroreLessicale() throw() {};
    const char* what() const throw()
    {
        string messaggio;
        messaggio = "non riconosco la parola " + parola;
        return messaggio.c_str();
    }
};
//funzione d'esempio che esegue un comando
void esegui(string comando)
```

```
{
    if(comandoNonRiconosciuto(comando))
        throw(ErroreLessicale(comando));
}

int main()
{
    try {
        esegui("Eccezzione"); //errore lessicale
    }
    catch(exception& e) { //cattura un'eccezione (polimorfismo)
        cout << e.what();
    }
    return 0;
}
```

Nota che in quest'esempio **ho passato l'eccezione come riferimento**: è una buona pratica da seguire sempre, dal momento che è indispensabile per implementare correttamente il comportamento polimorfo. Con un parametro normale, **e.what()** avrebbe richiamato la definizione di **what()** fornita dalla classe base **exception**, non da **ErroreLessicale**. Un altro gruppo di eccezioni è anche fornito dalla libreria standard: il mio consiglio è di evitare di utilizzare direttamente, ridefinire, o ereditare da queste. Meglio definire una **propria** gerarchia, evitando confusione nell'utilizzatore finale.

3.10 CONCLUSIONI

Le eccezioni sono il modo in cui il programmatore C++ può prevenire l'imprevedibile: un loro uso ragionato è quindi necessario in ogni applicazione che tenda alla robustezza, ancor meglio se combinato con tecniche formali di ingegnerizzazione del software, o approcci molto meno affidabili ma più "alla moda" come i test unita-

ri di XP [12, da leggere con una buona dose di spirito critico].
Come sempre: est modus in rebus.
La maggior parte delle istruzioni di un codice normale non è a rischio di generare un'eccezione: chi prevede blocchi try-catch ovunque non è diverso dall'ipocondriaco che si imbottisce di farmaci di ogni tipo per un banale raffreddore.
Queste sono le controindicazioni per il sovraddosaggio da eccezioni:

- **Il codice diventa difficile da leggere**. I blocchi try-catch prevedono un salto invisibile al primo errore incontrato: un lettore è quindi costretto ad analizzare a fondo il codice, per farsi un'idea corretta sull'esatto flusso dell'esecuzione.
- Dover gestire un'eccezione introduce un piccolo overhead: **tante eccezioni producono un grosso overhead**, che incide in maniera sempre più drammatica sulle prestazioni.
- **Chi usa la libreria perde completamente di vista qual è il rischio reale**, e quali sono i contorcimenti mentali di chi l'ha scritta.

C'è anche da dire che i programmatori tipici tendono a cronicizzare molto di più sul versante opposto: un falso senso di sicurezza per cui tutto va sempre nel migliore dei modi.
Con un po' di allenamento si arriva alla giusta via di mezzo.
Bisogna, infine, ricordarsi che quello delle eccezioni è un gioco cooperativo che prevede sempre due attori: chi le lancia l'eccezione e chi la gestisce.
Se è possibile realizzare il tutto da soli, ovvero se si ha un'informazione completa in mano, è assolutamente inutile e dannoso usare le eccezioni: è sufficiente gestire l'errore a livello locale con un bel, caro, vecchio **blocco if**.

PARADIGMA GENERICO

Il terzo dei modelli di programmazione che è possibile adottare con C++ è quello **generico**, il cui fondamento teorico è l'orientamento della programmazione agli algoritmi – per riferimenti dettagliati a questo sistema, puoi leggere [13]. Definito correttamente un algoritmo per un tipo di dato, è possibile scriverne un unico modello astratto da utilizzare per dati di ogni tipo attraverso un "polimorfismo parametrico" (cioè passando il tipo da usare, come argomento). Quest'approccio è quasi opposto a quello OOP, che cerca invece di partire dagli oggetti, evidenziandone i tratti comuni, in modo tale che possano essere racchiusi in un'unica classe base sulla quale si possa quindi scrivere l'algoritmo, demandando alle classi derivate i comportamenti particolari. Con la recente introduzione dei **generics** in linguaggi come C# e Java, il mercato mainstream ha fornito un'ulteriore prova empirica di quanto era già noto in ambiti più accademici ed elitari: il paradigma OOP è carente (in termini di flessibilità ed efficienza) nella gestione di alcune strutture irrinunciabili come collezioni, liste, stringhe, ed array associativi. Per questo motivo il C++ fa un uso massiccio della programmazione di tipo generico nella libreria standard, tramite il meccanismo dei **template**.

4.1 FUNZIONI TEMPLATE

4.1.1 ALGORITMI INDIPENDENTI DAI TIPI

Molti algoritmi formali sono indipendenti dai tipi utilizzati: un esempio semplice è lo scambio fra due variabili.
Scriviamo la funzione per due interi:

```
void scambia(int& a, int& b)
{
    int c = a;
    a = b;
```

```
    b = c;
}
```

Questo tipo di algoritmo è identico per ogni altro tipo primitivo (char, char*, long), e per ogni tipo composito che permetta la copia e l'assegnamento (struct Punto, class Frazione, etc…), motivo per cui per implementare questa soluzione in tutti questi tipi sarebbe necessario scrivere più volte lo stesso codice cambiando semplicemente il tipo.
Un'operazione del genere si può fare con un po' di organizzazione, mettendo la funzione in un file isolato e usando una qualche forma di automazione "trova e sostituisci…" per creare un nuovo overload per la funzione, sostituendo "int" con il tipo desiderato.
Ma agire così è sicuramente scomodo e limitato.

4.1.2 TEMPLATE

Il C++ offre un meccanismo molto più sofisticato e automatico per gestire questa situazione: dietro le quinte il compilatore attua da solo un'operazione di ricopiatura, ogni volta che serve, ma tanta tediosa complessità non appare all'utente finale.
Il modello base da cui derivano le "copie" prende il nome di **template**: come esempio di dichiarazione di un template possiamo provare a riportare in scrittura generica la versione specifica proposta nel paragrafo precedente:

```
template<class T> void scambia(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
}
```

Anteponendo "template<class T>" (o anche "template<typename

T>") all'intestazione della funzione, abbiamo dichiarato che T è un tipo generico, che può assumere qualsiasi forma. In questo modo, possiamo utilizzare T all'interno della funzione in maniera coerente.

```
int main()
{
    int i1=1, i2=2;
    scambia(i1,i2);
    cout << "i1 = " << i1 << " i2 = " << i2 << endl;

    Frazione f1(1,2), f2(2,3);
    scambia(f1, f2);
    cout << "f1 = " << f1 << " f2 = " << f2 << endl;

    return 0;
}
```

In questo codice d'esempio abbiamo utilizzato la funzione **scambia** con due tipi completamente diversi (int e Frazione).
Nota, peraltro, come in C++ le caratteristiche tipiche delle classi (come l'overloading degli operatori (<<)) si sposino senza problemi con la programmazione di tipo generico.
Dietro le scene il compilatore ha creato due versioni concrete di T: una per T=int, l'altra per T=Frazione: il raddoppiamento del codice sorgente (che pur sempre sussiste), però, non appare in alcun modo diretto al programmatore

4.1.3 PARAMETRI DEI TEMPLATE

Nel paragrafo precedente abbiamo usato un solo parametro, che abbiamo chiamato 'class T'. Un template può avere un numero arbitrario di argomenti, con qualsiasi nome e di qualsiasi tipo. Se non si conosce il tipo di un argomento, è possibile specificarlo come **class** (come abbiamo fatto con T) oppure con la parola chiave più "neu-

tra" **typename**, che è più esatta (in fondo, T può essere anche una struct, o un tipo primitivo), ma meno diffusa nella pratica comune, probabilmente perché di recente introduzione nello standard

4.1.4 TYPENAME

Il fatto che le classi passate come parametri siano generiche può portare ad alcune ambiguità nel codice.
Ad esempio, può succedere (fidati: nella libreria standard succede spesso) di volersi riferire ad un tipo interno ad una classe, come un typedef o una classe nidificata.
Un esempio può essere il seguente:

```
template<class T> void funzione(T& a)  //o typename T
{
    T::punto *p;
}
```

Ora, mentre per gli umani è alquanto evidente che questa riga indica una dichiarazione del puntatore p appartenente al sottotipo T::punto, il freddo compilatore (che giustamente segue le regole del linguaggio alla lettera) non avrà modo di stabilire se questa istruzione non indichi piuttosto una moltiplicazione fra il membro statico "punto" della classe T e un'ipotetica variabile globale di nome p.
In questo caso dobbiamo specificare che T::punto è un tipo. Per questo usiamo la parola typename.

```
typename T::punto *p;
```

Typename ha quindi una **doppia valenza**: da una parte può essere usato come (miglior) sinonimo di **class** per indicare un parametro di tipo, e dall'altra può essere utilizzato in quei casi in cui l'ambiguità renda necessario puntualizzare che ci si sta riferendo ad un tipo, e non ad un membro.

4.1.5 TEMPLATE E ISTANZE

Il paragrafo precedente evidenzia in modo chiaro che molte operazioni stabilite nei template assumono un significato definito solamente nella loro implementazione effettiva.
Questo implica che anche molti vincoli, problemi, ambiguità ed errori possono essere rilevati dal compilatore solo allorquando si faccia riferimento ad un'istanza particolare del template.
Ad esempio:

```
template<class T> T media(const T &a, const T &b)
{
    return (a+b)/2;
}
```

Questa funzione appare corretta e innocua allo sguardo umano, ma i primi due capitoli di questo libro ci hanno fornito abbastanza indicazioni per vederla con gli occhi di un compilatore! Molto probabilmente l'implementazione "dietro le quinte" della funzione **media** sarà qualcosa di simile:

```
T r1 = a.operator+(b);
T r2 = r1.operator/(2);   //oppure T r2 = r1.operator/(T(2));
return r2;
```

Ciò appare molto meno innocuo: innanzitutto occorre che il tipo T (ri)definisca l'operazione di addizione.
Poi occorre che in qualche modo sia definita o una divisione a intero, oppure una divisione a T e una conversione implicita da int a T (via costruttore, o per operatore di conversione).
Il compilatore è costretto ad assumere questi vincoli come impliciti, e verificarli puntualmente ad ogni istanza:

```
int main()
```

```
{
    Vettore a, b, c;
    c = media(a,b);     //molto probabilmente errore
}
```

L’istanza **media<Vettore>** utilizzata qui sopra è probabilmente un errore, a meno che la classe Vettore non definisca in maniera coerente l’addizione e la divisione a intero, rispettando i vincoli impliciti appena descritti.

4.2 SOVRACCARICAMENTO DI FUNZIONI TEMPLATE

Spesso esistono versioni generiche di un algoritmo, ma ne esistono anche di più rapide, per uno specifico tipo di dato.
Ad esempio, abbiamo visto nel paragrafo 2.11 del libro “Imparare C++” che l’algoritmo di scambio più rapido per gli interi è quello di xor: possiamo allora definire un sovraccaricamento di scambia() specifico per gli interi.

```
#include <iostream>
using namespace std;

template<class T> void scambia(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
    cout << “e’ stata usata la versione generica”;
}

void scambia(int& a, int& b)
{
```

```
    a ^= b;
    b ^= a;
    a ^= b;
    cout << "e' stata usata la versione int";
}

int main()
{
    int i1=1, i2=2;
    scambia(i1,i2);

    Frazione f1(1,2), f2(2,3);
    scambia(f1, f2);

    return 0;
}
```

Il risultato di questo codice sarà:

```
e' stata usata la versione int
e' stata usata la versione generica
```

Ciò dimostra che il compilatore è in grado di seguire una serie di regole per determinare quale sovraccaricamento sia più adatto per il tipo in questione.

4.3 CLASSI TEMPLATE

Introdotte le funzioni di tipo generico, potremmo applicare lo stesso principio anche alle classi.
A dire il vero, questo è il modo più comune di intendere i template, che viene utilizzato dalla libreria standard per quasi tutti i tipi che essa fornisce.

Un esempio immediato viene fornito dalla nostra classe Vettore, che abbiamo definito più volte nel corso di questo libro. Il problema peggiore di questa classe è che funziona solo con i dati di tipo intero.
E se avessimo bisogno di un vettore di char?
Dovremmo selezionare tutto, creare un altro file, copiare, incollare e sostituire da int a char.
Questa è esattamente la stessa premessa per la quale abbiamo definito le funzioni generiche.
La procedura per ottenere delle classi parametriche è esattamente la stessa di quella usata per le funzioni generiche:

```
#include <iostream>

using namespace std;

template<class T> class Vettore {
private:
    T* elementi;
    T grandezza;

public:
    Vettore(int g) : grandezza(g)
    {
        //crea i nuovi elementi
        elementi = new T[grandezza];

        //inizializza ogni elemento a zero
        for (int i=0; i<grandezza; i++)
            elementi[i] = 0;
    }

    T getGrandezza() {return grandezza;}
    T& operator[](int pos);
```

```
    //...
};
```

Questa trasformazione permette di utilizzare la classe Vettore su un **qualsiasi tipo**, sia esso primitivo o composito.
Ad esempio:

```
int main(void)
{
    Vettore<int>  v1(20);
    Vettore<char> v2(20);

    v1[0] = v2[0] = 64;

    cout << v1[0] << endl;
    cout << v2[0] << endl;

    return 0;
}

```

```
64
@
```

4.4 PARAMETRI MULTIPLI E COSTANTI

Finora abbiamo usato un parametro solo, e abbiamo usato delle classi come parametro: né l'uno né l'altro sono vincoli da rispettare.
È possibile usare un numero arbitrario di parametri in un template, e anche dei tipi precisi, purché vengano passate delle costanti.
Ad esempio, potremmo scrivere delle versioni di Vettore che faccia-

no a meno della memoria dinamica, per quelle situazioni in cui sappiamo già quanti elementi ci servono, in questo modo:

```
template<class T, int g> class Vettore {
private:
    T elementi[g];
public:
    Vettore()
    {

        //inizializza ogni elemento a zero
        for (int i=0; i<g; i++)
            elementi[i] = 0;
    }
    T getGrandezza() {return g;}
    T& operator[](int pos);
  //...
};

int main(void)
{
    Vettore<int, 20> v1;
    Vettore<char, 20> v2;
    v1[0] = v2[0] = 64;

    cout << v1[0] << endl;
    cout << v2[0] << endl;

    return 0;
}
```

Questo sistema è nettamente migliore quando si ha a che fare con degli array di cui si conosce a priori la dimensione: dal momento che

usa memoria statica non ha problemi di distruttori, e simili.
Ricorda bene, però, che quando si usa un template, come ad esempio **Vettore<int, 20>**, viene creata una nuova classe dietro le scene, in cui 'int g' viene sostituito con la costante passata (cioè 20).
Per questo motivo è impossibile richiamare un template con dei valori non costanti, ad esempio **Vettore<int, a>**.

4.5 PARAMETRI PREDEFINITI

Come le funzioni, anche i template di funzioni possono essere sovraccaricati o i loro parametri possono essere resi di default.
Ad esempio, nell'implementazione del template di classe Vettore descritto nel paragrafo precedente, è possibile dichiarare come predefinito il parametro g:

```
template<class T, int g=10> class Vettore {
    //…
} ;
```

Questo permette di istanziare un Vettore senza dichiararne la dimensione: verrà assunto implicitamente il valore di default (10, in questo caso):

```
int main()
{   Vettore<Cane*> cani;  //Vettore <Cane*, 10>
    Cane* cane = cani[20];          //errore: indice fuori dai margini
    return 0;
}
```

4.6 SPECIALIZZAZIONE DEI TEMPLATE

I template delle classi non si possono sovraccaricare.

Ma si possono **specializzare**, ovverosia è possibile ridefinire il template per un particolare tipo di parametro. Ad esempio, la classe **Vettore** è particolarmente dispendiosa nel caso di dati di tipo bool (la libreria standard prevede il simil-contenitore **bitset** per questo): per identificare 20 elementi di tipo bool occorrono 20 bytes.
Potremmo scriverne una versione specializzata capace di usare n bits.

```
Template<> class Vettore<bool>
{
    char* bytes; //base di bytes
    bool& getBit(n); //preleva il bit in posizione n
public:
    bool& operator[](int pos)
    {
        if (pos < grandezza && pos >= 0)
            return getBit(pos);
        else {
            //..
        }
    }
};
```

Una volta definito il metodo **getBit()** (farlo in questa sede sarebbe fuori luogo e richiederebbe troppo spazio), avremo una **specializzazione** funzionante della classe Vettore per parametri di tipo **bool**.

4.7 CONCLUSIONI

Al di là delle Guerre Sante, è ormai evidente che il paradigma generico rappresenta un'ottima integrazione di quello ad oggetti, dal momento che permette di ricorrere al polimorfismo parametrico a tempo di compilazione, evitando così il ricorso a soluzioni intrinsecamente molto meno sicure ed efficienti, come il downcasting o il ty-

pe punning a runtime.
Come abbiamo visto, i template non sono esenti da difetti: il codice "invisibile" prende spazio, e ciò porta inevitabilmente alla lievitazione dell'eseguibile generato dal compilatore; inoltre è vero che gli errori e le ambiguità del codice possono essere come sempre risolti grazie ai messaggi del compilatore, ma ciò avviene di volta in volta, quando il template è stato istanziato con dei parametri concreti: ciò impone sempre di conoscere in anticipo i requisiti che un tipo deve soddisfare per istanziare un template, leggendosi la documentazione - sarebbe utile (ma poco fattibile) avere un meccanismo di "specifica dei template", al pari delle eccezioni (vedi paragrafo 3.7).
Come al solito, saper scegliere quando usare il modello generico invece dell'OOP è una questione d'esperienza, e anche di gusti e preferenze.
Come vedremo, la libreria standard fa grande uso di questo paradigma al suo interno sia per quanto riguarda i template a classe (contenitori, stringhe, stream…), sia per quanto riguarda i template a funzione (algoritmi parametrici): creare classi o algoritmi personalizzati per estendere le funzionalità della libreria standard può essere un ottimo modo per acquisire familiarità con questo potente strumento.

LIBRERIA E CONTENITORI STANDARD

Un buon programmatore C++ conosce bene la libreria standard, in tutte le sue sfaccettature.
Non è certo un compito semplice, ma aiuta molto: reinventare la ruota non è mai divertente, né semplice, né comodo, né sicuro, né performante.
Grazie alle classi e ai template, tutto questo viene fornito in maniera semplicissima dalla libreria standard, che fa di tutto per renderci la vita più semplice e sicura. Dalle origini ad oggi la libreria standard si è ampliata moltissimo, rielaborando la libreria C standard e definendo delle nuove classi e funzioni, ed è certo che l'obiettivo futuro del comitato C++ sarà proprio quello di garantirne uno sviluppo ulteriore. Esaurire l'argomento è impossibile, dal momento che richiederebbe tutto un libro a parte ([1 e 10] sono ottimi riferimenti).
Puoi usare questo breve capitolo come specchietto panoramico sull'argomento, i capitoli che seguono per un approfondimento sugli argomenti fondamentali, e ai riferimenti bibliografici per le sezioni che in sole 160 pagine è impossibile ricoprire.

5.1 LIBRERIA C

La libreria standard riprende quasi completamente la libreria C, con delle piccole variazioni per omogeneizzarla alle novità offerte dal linguaggio.
Per indicare che un header fa parte della libreria C è previsto l'uso di una "c" iniziale; inoltre tutti gli header della libreria standard non hanno estensione.
Gli header standard vanno sempre preferiti alle vecchie versioni (ad esempio, è preferibile scrivere #include<ctime> rispetto a #include <time.h>) Questo è l'elenco degli header più importanti, con una spiegazione del loro scopo:

Header	Funzionalità fornite:
<cmath>	Funzioni matematiche e trigonometriche (ad esempio sin(double), floor(double)…) e costanti (ad esempio PI).
<cstdlib>	Funzioni di vario tipo: dalla matematica alla generazione di numeri casuali, passando per alcune funzioni per le strignhe, di ordinamento, e molto altro…
<ctime>	Funzioni per l'ora, la data, e la misurazione del tempo
<cstring>, <cwchar>	Funzioni per la manipolazione delle stringhe C-like (a terminatore nullo), e dei dei caratteri estesi.
<cstdio>	Funzioni per la scrittura e la lettura da i/o. Fanno parte di questa famiglia i vari printf, scanf, e derivati…

Tabella 5.1: header principali legati alla libreria C

Molte delle funzioni fornite da queste librerie trovano un corrispettivo migliore e più immediato in funzioni tipicamente C++: ad esempio <cstdio> è in larga misura sostituibile utilizzando <iostream>, anche se non completamente.
In situazioni del genere è fortemente consigliato scegliere la via C++ (ad esempio, usare cout al posto di printf, cin al posto di scanf, strstream al posto di itoa, etc…).

5.2 CONTENITORI (CAPITOLO 5)

La libreria standard del C++ prevede una serie di contenitori generici basati su template. Lo schema dei template può risultare piuttosto complesso, utilizzeremo qualche tabella riassuntiva per poterci orientare facilmente. La tabella 5.3 illustra gli header principali collegati ai vari contenitori:

Header	Struttura	Classificazione	Paragrafo
<vector>	Array	Sequenza	5.9
<list>	Lista collegata	Sequenza	5.10.1
<deque>	Coda a due capi	Sequenza	5.11.1
<queue>	Coda	Adattatore	5.11.2
<queue>	Coda con priorità	Adattatore	5.11.2
<stack>	Pila	Adattatore	5.11.2
<map>	Array associativo	Associativo	5.12.1
<set>	Insieme	Associativo	5.12.2

Tabella 5.2: header principali legati ai contenitori standard

Quando le prestazioni non sono un problema primario, è sempre meglio utilizzare questi contenitori rispetto alle strutture più primitive (array, linked list, etc…) e meno controllate.

5.3 ALGORITMI E OGGETTI FUNZIONE (CAPITOLO 5)

Sui contenitori standard è possibile applicare una buona serie di algoritmi, forniti principalmente dagli header <algorithm> e <numeric>.
Questi, combinati all'uso degli iteratori (header <iterator>) e degli oggetti funzione (header <functional>) offrono la possibilità di operare in maniera generica e personalizzata.

5.4 STRINGHE (CAPITOLO 6)

La libreria standard fornisce una classe basic_string<> per la dichiarazione di stringhe generiche di un certo tipo di carattere, rappresentato dalla classe char_traits. Possiamo usare diversi header per gestire questa libreria
La tabella 5.3 illustra gli header principali collegati a stringhe e caratteri:

Header	Funzionalità offerte:	Paragrafo
<string>	Stringhe generiche	6.2
<cctype>	Caratteri	-
<cwtype>	Caratteri estesi	-

Tabella 5.3: header principali legati alle stringhe

Soprattutto nel caso delle stringhe, vale spesso la pena di preferire l'uso delle classi fornite dalla libreria (come base_string<> e string), piuttosto che complicarsi la vita con le stringhe in stile C (come char* e wchar_t*).

5.5 STREAM (CAPITOLO 6)

Gli stream sono l'astrazione che il C++ fornisce per la gestione del flusso su vari devices.
La tabella 5.4 illustra gli header principali collegati agli stream:

Header	Funzionalità offerte:	Paragrafo
<ios>	Stream di base	6.7
<istream>	Stream di input	6.3.1
<ostream>	Stream di output	6.3.2
<iostream>	Stream di input e di output	6.4
<sstream>	Stream su stringhe	6.5
<fstream>	Stream su files	6.6
<streambuf>	Stream bufferizzati	-
<iomanip>	Manipolatori	6.7

Tabella 5.4: header principali legati agli stream

5.6 FUNZIONALITÀ MATEMATICHE

La libreria standard offre anche supporto per alcune computazioni di carattere tipicamente matematico: l'header <complex> fornisce una classe per la rappresentazione dei numeri complessi e delle operazioni

tipiche ad essi correlate.
Mentre l'header <valarray> permette di gestire vettori numerici, e un insieme di operazioni tipiche del calcolo matriciale.

5.7 SUPPORTO AL LINGUAGGIO E DIAGNOSTICA

La libreria standard copre anche un gran numero di altre funzionalità (più o meno secondarie), che sarebbe inutile elencare separatamente o nel dettaglio, dato lo scopo di questo libro.
La tabella 5.5 ne elenca solamente alcune:

Header	Funzionalità offerte:
<limits>, <typeinfo>, <exception>…	Supporto al linguaggio
<cassert>, <cerrno>…	Asserzioni e diagnostic
<locale>, <clocale>	Gestione differenze culturali

Tabella 5.5: alcuni header di funzionalità miscellanee

I CONTENITORI STANDARD

Una delle funzionalità più note ed utilizzate della libreria standard sono senza dubbio i contenitori: un insieme di classi parametriche in grado di "contenere" altri oggetti (proprio come la nostra classe Vettore).
A differenza di molti altri linguaggi, nella libreria standard del C++ si è voluta dare un'enfasi particolare alle prestazioni, alla semplicità d'uso e alla possibilità di definire contenitori intercambiabili – certo, non compiti facili. La base dei contenitori C++ è il lavoro di Stepanov e Lee sulla Standard Template Library [10]: una ricerca svolta per determinare la maniera più efficiente possibile per utilizzare contenitori generici in C++.
Per questa ragione questa parte della libreria viene comunemente (anche se non ufficialmente) chiamata **"framework STL"**.

5.8 OBIETTIVI DEI CONTENITORI C++

5.8.1 I PROBLEMI DELL'APPROCCIO IN STILE SMALLTALK

Normalmente, quando si progetta un insieme di classi comuni si segue l'approccio tipicamente orientato agli oggetti in stile Smalltalk: si identificano le necessità comuni delle classi, le si raggruppa in una singola interfaccia virtuale che funga da classe base e che debba essere ereditata da ogni singola classe.
Gli antenati dei contenitori C++ odierni seguivano questo schema: offrivano un "contenitore virtuale" che esponeva delle funzioni (ad esempio, l'operatore [], l'inserimento, etc…), e che poi doveva essere ereditato dai contenitori concreti (come vector).
Ma questa struttura si è rivelata ben presto inadeguata.
I problemi erano due: innanzitutto, le classi e le funzioni virtuali, come abbiamo visto nel capitolo 2, occupano più spazio in memoria a runtime e sono meno efficienti, a causa della presenza dei vptr e del meccanismo di chiamata a vtable; il secondo problema è che effettivamente ogni contenitore ha una struttura interna particolare e differente dagli altri, pertanto non tutti sono in grado di garantire sempre le definizione dello stesso insieme di operazioni: una lista, ad esempio, non potrà esporre l'operatore per l'accesso casuale ("[]").
Dover prevedere un gran numero di eccezioni di tipo "funzione non supportata" per un contenitore non è comodo, e soprattutto non è efficiente.
Tutta l'architettura dei contenitori C++, invece, è progettata per offrire semplicità e buone prestazioni.

5.8.2 L'APPROCCIO STRUTTURALE IN STILE STL

Come fa la libreria standard a superare i problemi esposti nel capitolo precedente?

L’approccio è il seguente: si definisce sempre un insieme tipico di operazioni svolte da un contenitore tipico, tuttavia **non** ne si fa una classe base, ma lo si tiene come riferimento di progettazione.
I contenitori sono **specifici**, ovverosia operano su un solo tipo di dati, astratto nell’implementazione tramite il meccanismo dei template; in questo modo si evita completamente il problema di dover ricorrere a downcasting dinamici (lenti) per verificare ogni volta il tipo degli oggetti passati.
Così, ad esempio, in ogni classe vengono implementati dei metodi per restituire un tipo di **iteratore** compatibile con la struttura dati del contenitore, che permetta di ottenerne un puntatore ad un elemento e di navigarne la struttura.
Ciò risolve nella maniera migliore i vincoli prestazionali e di flessibilità. Inoltre è molto semplice creare nuovi contenitori per ereditarietà o specializzazione di quelli esistenti.
La tabella 5.2 illustra i contenitori fondamentali secondo una classificazione in tre tipi: **le sequenze** sono quei contenitori che mantengono gli elementi in un ordine preciso; gli **adattatori** sono contenitori “costruiti sopra” altri contenitori, gli **associativi** sono quelli nei quali la posizione degli elementi ha un’importanza marginale (o non ne ha affatto), e in cui questi si richiamano, invece, attraverso una chiave a loro associata.
Tutti questi contenitori appartengono al namespace std e possono essere utilizzati includendo il relativo file header.

5.9 UN ESEMPIO DI CONTENITORE: VECTOR

Vector è il contenitore più noto, per il fatto che somiglia molto ad un comune array. In realtà è molto più flessibile e dovrebbe essere sempre preferito all’uso di un vettore primitivo, per tutti i motivi che ho già esposto nel libro “Imparare C++”.
Si può dire che vector sia la “bella copia” della nostra classe Vetto-

re: ha lo stesso scopo ma è molto più efficiente, riutilizzabile, versatile, e fornisce molte più operazioni.
Qui di seguito mostro le principali, che fungeranno da riferimento anche per gli altri tipi di contenitore.

5.9.1 COSTRUZIONE

Un vettore opera su un blocco contiguo di memoria.
Per questo l'allocazione per gli elementi viene effettuata subito, e il numero di elementi può essere indicato al momento della costruzione, ed è facoltativamente possibile indicare un "valore d'inizializzazione" al quale porre ogni elemento.

```
//uso di vari costruttori di Vector
#include <vector>
using namespace std;

int main()
{
    vector<long> vuoto;            //nessun elemento
    vector<Cane> dieciCani(10);    //dieci elementi di tipo Cane
    vector<char> dieciEnne(10, 'n');          //dieci caratteri di valore 'n'

    return 0;
};
```

5.9.2 ACCESSO AD UN ELEMENTO

Oltre che mediante un iteratore (vedi 5.10.2), vector permette l'accesso via indice attraverso l'operatore [].
Il codice riportato qui sotto assegna ai dieci elementi del vettore "numeri" i primi dieci numeri interi.

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
```

```
    numeri[i] = i;
```

Abbiamo visto dalla nostra implementazione di Vettore che questo genere di classi può generare facilmente un errore di tipo “indice fuori dai margini”, e che è consigliabile implementare un’eccezione apposita.
L’operatore [] di vector non lo fa, per motivi prestazionali, pertanto è consigliabile chiamarlo solo quando si è certi di essere all’interno del range consentito.
In caso contrario, è possibile richiamare la funzione **at()**, che è identica all’operatore di accesso via indice, ma che implementa un’eccezione di tipo out_of_range:

```
vector<int> numeri(10);
try
{
    numeri.at(20) = 1; //out_of_range
}
catch(out_of_range)
{
    cout << “L’indice è fuori dai margini”;
}
```

L’indice è fuori dai margini

I metodi **front()** e **back()** restituiscono un riferimento rispettivamente al **primo** e all’**ultimo** elemento del vettore:

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
    numeri[i] = i;
cout    << “I numeri vanno da ”
<< numeri.front();
```

```
<< "a" << numeri.back();
```

```
I numeri vanno da 0 a 9
```

5.9.3 INSERIMENTO E RIMOZIONE DI ELEMENTI

A differenza della nostra classe Vettore, vector non si fa problemi ad inserire nuovi elementi al suo interno (**come** lo permetta dietro le scene è discusso nel prossimo paragrafo). Il metodo **push_back**(const T&) inserisce un elemento alla fine del vettore (cioè, dopo l'ultimo):

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
    numeri[i] = i;
numeri.push_back(1000);
cout << "L'ultimo elemento e' " << numeri.back();
```

```
L'ultimo elemento e' 1000
```

Analogamente, il metodo **pop_back()** permette di eliminare l'ultimo elemento del vettore.

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
    numeri[i] = i;
numeri.pop_back();
cout << "L'ultimo elemento e' " << numeri.back();
```

```
L'ultimo elemento e' 8
```

Vector permette anche l'inserimento di un elemento in una posizione precisa, di modo che gli altri "scalino a destra" per fare spazio al nuovo arrivato.

Ciò è permesso grazie al metodo **insert**, che accetta due parametri: il primo è un iteratore all'elemento da sostituire, il secondo è il valore del nuovo elemento.
Qui mostro come inserire un elemento all'inizio di vector, in una sorta di push_front():

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
    numeri[i] = i;
numeri.insert(numeri.begin(), 1000);
cout << "Il primo elemento e' " << numeri.front();
```

```
Il primo elemento e' 1000
```

Analogamente, è possibile **eliminare** un elemento che non si desidera più, facendo "scalare a sinistra" tutti gli altri a coprire il vuoto. Ciò è permesso grazie al metodo **erase**, che accetta come parametro un iteratore all'elemento da rimuovere.
Qui mostro come rimuovere un elemento all'inizio del vector, in una sorta di pop_front():

```
vector<int> numeri(10);
for (int i=0; i<10; i++)
    numeri[i] = i;
numeri.erase(numeri.begin());
cout << "Il primo elemento e' " << numeri.front();
```

```
Il primo elemento e' 1
```

Infine, è possibile **eliminare tutto** il contenuto del vettore, utilizzando il metodo **clear()**.

```
vector<int> numeri(10);
```

```
for (int i=0; i<10; i++)
    numeri[i] = i;
numeri.clear();
numeri.push_back(1000);
cout << "Il primo elemento e' " << numeri.front();
```

```
Il primo elemento e' 1000
```

5.9.4 DIMENSIONE E RILOCAZIONE

Come fa il vettore a ridimensionarsi da solo?
Per scoprirlo basta pensare a come è strutturato un vector, cioè in maniera non dissimile dalla classe Vettore.
C'è un insieme di elementi di un tipo, che vengono allocati dinamicamente dal contenitore al momento della costruzione (vedi 5.9.1).
Poniamo caso che un vettore di interi contenga 10 elementi.
E' molto facile per una classe vector **eliminare l'elemento in coda** (ovverosia effettuare una pop_back()): basta decrementare la dimensione, ovverosia "far finta" che l'ultimo elemento non sia più accessibile, e richiamarne il distruttore.
Analogamente è relativamente semplice **eliminare un elemento qualsiasi** (ovverosia effettuare una **erase**()): basta decrementare la dimensione, distruggere l'elemento da eliminare, e far scalare di posto gli altri. Proprio quest'ultimo passaggio rende l'operazione più dispendiosa di un semplice pop_back(), fatto da cui consegue che è sempre più efficiente utilizzare un vector come se fosse una pila. In entrambi i casi ho sempre fatto riferimento ad una variabile "dimensione", che nella nostra classe Vettore prendeva il nome di "grandezza".
Nei contenitori STL questa variabile viene chiamata **size**, ed è accessibile attraverso la funzione size() che restituisce il numero totale degli elementi contenuti, e attraverso **resize**(nuovoNumeroDiElementi) per il ridimensionamento.

```
vector<int> numeri(10);
```

```
cout << "il vettore numeri ha " << numeri.size() << " elementi";
vector.resize(9);
cout << "il vettore numeri ha " << numeri.size() << " elementi";
```

```
il vettore numeri ha 10 elementi
il vettore numeri ha 9 elementi
```

Il problema, però, sta nell'operazione opposta: fornire un'implementazione dei metodi **push_back** e **insert**.
Qui per la classe vector non è possibile "far finta" di nulla, incrementando semplicemente la variabile size, dal momento che gli elementi finirebbero a puntare a memoria non ancora allocata.
Ecco, quindi, che vector ha due alternative: la prima è quella di "espandersi" occupando anche le celle successive.
Ciò, però, non è possibile se queste sono occupate da qualche altro oggetto. In questo caso (che avviene di frequente), a vector non rimane che fare le valigie e traslocare in un'area di memoria più capiente, che sia in grado di garantire lo spazio anche per il nuovo elemento.
Quest'operazione è **molto** dispendiosa in termini di tempo, dal momento che gli elementi devono effettivamente essere ricopiati da una parte all'altra della heap, il che significa altre copie di ciascun oggetto per gli elementi di destinazione, e altre distruzioni di ciascun oggetto per gli elementi di partenza.

```
vector<int> numeri(10);
    cout << "il primo elemento e' allocato qui: "
<< &numeri[0] << endl;
    numeri.resize(20);
    cout << "ora il primo elemento e' allocato qui: " << &numeri[0]
```

```
il primo elemento e' allocato qui: 00355AA0
ora il primo elemento e' allocato qui: 00355BA0
```

Ecco perché nei casi in cui le prestazioni siano un problema, è sempre meglio evitare numerose operazioni di inserimento in un vector. Se hai necessità di realizzare un array incrementale e conosci già la dimensione massima alla quale questo potrà arrivare, puoi aggirare l'ostacolo facendo allocare preventivamente un area di memoria adeguata ad ospitare **tutti** gli elementi possibili.
La funzione **reserve**(numeroElementi) ha proprio questa finalità:

```
vector<int> numeri(10);
numeri.reserve(20);
cout << "il primo elemento e' allocato qui: " << &numeri[0] << endl;
numeri.resize(20);
cout << "il primo elemento e' sempre allocato qui: " << &numeri[0]
```

```
il primo elemento e' allocato qui: 00355B08
il primo elemento e' sempre allocato qui: 00355B08
```

Come si evince dall'esempio, il vantaggio di usare reserve è che si ottiene la garanzia che non ci sarà bisogno di rilocare il vettore, senza per questo fissare per forza la dimensione degli elementi: size restituisce comunque il numero degli elementi realmente esistenti, non di quelli "riservati".
Se si vuole avere il numero degli elementi totali (cioè gli elementi di size() più quelli ancora non utilizzati) ci si può servire della funzione **capacity**().

```
vector<int> numeri(60);
numeri.reserve(100);

cout << "Il vettore numeri ha " << numeri.size() << " elementi.";
cout << "E' stata riservata memoria per " << numeri.capacity()
<< " elementi.";
cout    << "E' possibile aggiungere ancora "
```

```
    << numeri.capacity() – numeri.size() << " elementi.";
```

```
Il vettore ha 60 elementi
E' stata riservata memoria per 100 elementi
E' possibile aggiungere ancora 40 elementi
```

Bada che l'ultima affermazione non significa che dopo la quarantesima aggiunta non sarà più possibile eseguire un'altra operazione di inserimento o di push.
Semplicemente, se ci si spinge più in là di altri 40 elementi, il vettore sarà probabilmente rilocato ad ogni inserimento (oppure bisognerà fare un altro reserve per assicurarsi un altro po' di memoria).

5.9.5 DESCRITTORI DI VECTOR

Ogni contenitore definisce una serie di typedef al suo interno, che permettono di avere accesso al tipo degli elementi, dell'allocatore, degli iteratori, e di altre caratteristiche.
Ad esempio:

```
vector<int> numeri;
//vector<int> contiene degli int?
typeid(vector<int>::value_type) == typeid(int);  //vero
//l'allocatore di vector<int> è un allocatore di int?
typeid(vector<int>::allocator_type) == typeid(allocator<int>);
//vero
//eccetera
```

I tipi definiti da ogni contenitore sono molti: dalla dimensione di un elemento, al tipo di un riferimento. I più comunemente utilizzati sono quelli che definiscono gli iteratori (vedi 5.10.2)

5.9.6 OPERATORI

Tutti i contenitori definiscono una serie di operatori specializzati.

Vector, ad esempio permette di costruire un vettore a partire da un altro, grazie all'operatore di assegnamento e al costruttore per copia.

```
vector<int> numeri(10, 100);
vector<int> copia = numeri;
cout << copia[5]; //100
```

Altri operatori interessanti possono essere quelli di confronto: vector li definisce per stabilire un confronto lessicografico fra due vettori (in pratica, si valutano i confronti elemento-per-elemento, come si fa comunemente per le stringhe).

```
vector<int> numeri1, numeri2;
numeri1.push_back(1); numeri1.push_back(5); numeri1.push_back(7);
numeri2.push_back(1); numeri2.push_back(5); numeri2.push_back(6);
// {1,2,7} è maggiore di {1,2,6}?
(numeri1 > numeri2); //vero
```

5.10 LIST E ITERATORI

5.10.1 IL CONTENITORE LIST

Tutti i contenitori seguono più o meno la struttura già spiegata per vector, implementandone le stesse funzioni e caratteristiche, anche se possono presentare delle differenze dovute alla natura stessa della struttura dati. Una **lista**, per esempio, è un contenitore in cui ogni elemento punta al successivo e al precedente (vedi figura 5.41a). Ciò permette di risolvere il problema del cambiamento di dimensioni: ogni volta che si vuole inserire un nuovo elemento (non importa in che posizione), basta semplicemente farlo puntare dagli elementi contigui, come illustra la figura 4.1b. **List** si usa quindi in tutti quei casi in cui il contenitore è soggetto a continui incrementi e rimozio-

ni (indifferentemente in testa, in coda e in mezzo), in quanto queste operazioni sono immediate e non è mai recessario rilocare la memoria, o "far scalare" gli elementi negli inserimenti. Per contro, però, dal momento che ogni elemento risiede in un indirizzo fisico scorrelato dagli altri, risulta impossibile fornire un accesso via indice, pertanto, come vedremo, la selezione di un elemento in una data posizione ha una complessità computazionale maggiore.
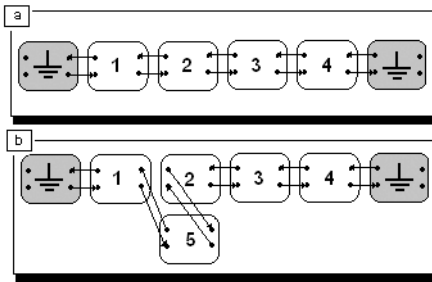
Figura 5.4.1: a) Lista collegata con doppia sentinella b) Inserimento dell'elemento 5 in seconda posizione

5.10.2 ITERATORI

Come si fa ad accedere, quindi, ad un elemento di list in una data posizione?

L'unica via è quella di arrivarci partendo dal primo elemento (o dall'ultimo) e andando in avanti (o all'indietro) di tante posizioni quanto desiderato.

Ciò implica che debbano essere offerte due funzionalità: la prima è di ottenere un oggetto che sappia muoversi in avanti e all'indietro fra i vari elementi di un contenitore, e saper dereferenziare quello puntato - ciò è fornito dagli **iteratori**.

La seconda funzionalità è che i vari contenitori espongano dei **metodi che restituiscano iteratori** al primo e all'ultimo elemento – ciò è fornito rispettivamente dai metodi **begin()** (che punta al pri-

mo elemento) e **end()** (che punta a quello **dopo** l'ultimo).

```
list<int> interi(20);    //crea una lista di 20 interi
list<int>::iterator i;    //iteratore in avanti

for (i=interi.begin(); i != interi.end(); i++) {
    *i = 1;    //dereferenziazione di come l-value
    cout << *i;    //dereferenziazione di come r-value
}
```

Da questo stralcio di codice è possibile capire molte cose: innanzitutto, che il tipo di un iteratore, viene esposto dal contenitore stesso tramite un typedef descrittore.
Ne esistono di diversi tipi: possono essere in normali o costanti (nel qual caso non si potranno dereferenziare come l-value), in avanti, o al rovescio.
In quest'ultimo caso un incremento farà andare l'iteratore all'indietro e un decremento in avanti; per ottenere un iteratore al rovescio, è necessario chiamare **rbegin()** e **rend()**.
L'unione di queste caratteristiche porta a quattro tipi di iteratori distinti:

list<T>:: iterator: Iteratore in avanti non-costante
list<T>:: const_iterator: Iteratore in avanti costante
list<T>:: reverse_iterator: Iteratore al rovescio non-costante
list<T>:: const_reverse_iterator: Iteratore al rovescio costante

NOTA
Queste categorie descrivono solo alcune delle caratteristiche che un iteratore può assumere, ma ne esistono molte altre. Gli iteratori possono essere classificati come di ingresso o di uscita, ad accesso casuale, ed esistono molti tipi specializzati (come gli inseritori).

Un'altra cosa che è evidente dal codice qui sopra riportato è che si può far avanzare l'iteratore di una posizione semplicemente incrementandolo o decrementandolo.
Questa è una caratteristica supportata da tutti gli iteratori, mentre **non** lo è l'incremento multiplo, ovvero l'uso dell'operatore += o -=.
Ad esempio, per una lista il codice seguente è scorretto:

```
for (i=interi.begin(); i != interi.end(); i+=2)  //errore
```

Altrettanto scorretta è l'assunzione che tutti gli elementi dell'iteratore siano in ordine crescente.
Per una lista, ad esempio, ciò non è vero, dal momento che la posizione degli elementi in memoria è del tutto scorrelata.
Per questo il ciclo for non prevede la scrittura (sbagliata):

```
for(i=interi.begin(); i < interi.end(); i++) //errore
```

Questa scrittura non si comporterà in maniera corretta, perché verifica se l'iteratore è **minore** di quello finale: un test che probabilmente non sarà neanche supportato dal compilatore.
Bada anche al fatto che l'elemento puntato da **end()** non è l'ultimo, ma un elemento **sentinella**, messo apposta per marcare la fine della sequenza (un po' come il carattere nullo nelle stringhe C).

```
cout << *interi.end(); //errore a runtime
```

L'istruzione riportata qui sopra darà probabilmente un errore a runtime perché dereferenzia l'elemento end(), che non punta a nessun elemento concreto.
Un'ultima osservazione che è possibile trarre da questi esempi è che **l'iteratore può essere trattato come un puntatore** all'elemento, pertanto una dereferenziazione ne permetterà l'accesso (in lettura e scrittura, se questa è permessa).

Poiché, però, un iteratore ha già i suoi metodi e i suoi membri, ogni operazione eseguita **senza** dereferenziarlo sarà intesa come rivolta all'iteratore, ad esempio:

```
list<Frazione> frazioni;
frazioni.push_back(Frazione(5,5)); inserisce la Frazione 2/5
list<int>::iterator i = interi.begin();

cout << i;    //sbagliato: si sta stampando l'iteratore
cout << *i;  //giusto: 5/5
i.semplifica()       //sbagliato: si sta "semplificando" l'iteratore
i->semplifica()    //giusto: 1/1
```

5.11 DEQUE E GLI ADATTATORI

5.11.1 DEQUE

La terza sequenza è **deque** (contrazione di double-ended-queue, pronuncia **dèk**): un buon compromesso fra vector e list.
Si tratta di una "coda a due capi", che ha l'efficienza di list per gli inserimenti in testa e in coda, e quella di vector per l'accesso via indice.
Se non si ha bisogno di un contenitore efficiente negli inserimenti centrali, dunque, una deque si rivela spesso la soluzione migliore.

5.11.2 ADATTATORI: STACK, QUEUE E PRIORITY_QUEUE

Una volta creato un contenitore, è possibile costruirvi sopra facilmente dei derivati, chiamati **adattatori**.
Gli adattatori non operano per ereditarietà, bensì per composizione: dichiarano il contenitore su cui si basano come membro privato o protetto, e forniscono al programmatore un'interfaccia che agisce su di esso per delega.

Gli adattatori possono limitare le funzionalità di un contenitore, o fornire delle funzionalità nuove: un esempio tipico di adattatore “riduttore d’interfaccia” è **stack**.
Si tratta di una “deque limitata”, che permette soltanto la rimozione (**pop**), l’inserimento (**push**) e la lettura (**top**) degli elementi in coda: ne risulta un contenitore compatto e semplice da usare.

```
stack<int> numeri;
numeri.push(1); numeri.push(2);   //numeri = {1,2}
cout << numeri.top();       //2
numeri.pop();               //numeri = {1}
numeri.pop();               //numeri = {}
```

Analogamente, **queue** (coda, pronuncia kiù) è un altro adatatore basato su deque, nel quale è possibile leggere gli elementi in testa (**front**) e in coda (**back**), rimuovere l’elemento in testa (**pop**), e aggiungere l’elemento in coda (**push**).

```
queue<int> numeri;
numeri.push(1); numeri.push(2)    //numeri = {1,2}
cout << numeri.front();           //1
cout << numeri.back();            //2
numeri.pop();               //numeri = {2}
numeri.pop();               //numeri = {}
```

Infine, una **priority_queue** è una coda in cui è possibile solo rimuovere l’elemento in testa (**pop**), aggiungere un elemento in coda (**push**) e leggere l’elemento in testa (**top**), proprio come nello stack.
La particolarità di questo contenitore, però, è che è possibile assegnare a ciascun elemento un indice di priorità.
Possiamo provare ad utilizzare priority_queue per simulare cosa succederebbe se, alle casse del supermercato, passassero prima i clien-

ti con meno articoli.
Innanzitutto definiamo la classe Cliente:

```
class Cliente
{
public:
    string      nome;
    int   articoli;

    Cliente(string n, int a) : nome(n), articoli(a) {};
    bool operator<(const Cliente& b) const {return articoli > b.articoli;}

};
```

Per indicare la priorità bisogna sovraccaricare l'operatore <.
In questo caso, un cliente ha meno priorità di un altro se ha meno articoli.
Ora vediamo come creare una coda con priorità:

```
priority_queue<Cliente> fila;

fila.push(Cliente("Tizio", 20));          //Tizio ha 20 articoli
fila.push(Cliente("Caio", 10));           //Caio ha 10 articoli
fila.push(Cliente("Sempronio", 10));      //Sempronio ha 10 articoli

cout << fila.top().nome << endl; fila.pop(); //Caio/Sempronio
cout << fila.top().nome << endl; fila.pop(); //Caio/Sempronio
cout << fila.top().nome << endl; fila.pop(); //Tizio
```

Caio e Sempronio si troveranno ai primi posti della fila, anche se non possiamo essere certi della posizione esatta perché il comportamento della coda è a discrezione del compilatore in caso di pareggio.

Sappiamo per certo, però, che Tizio, gravato dai suoi 20 articoli, sarà servito per terzo.

5.12 CONTENITORI ASSOCIATIVI

5.12.1 MAP E ITERATORI A COPPIE

I contenitori associativi sono la versione comoda e pulita che il C++ offre delle famigerate strutture “ad albero”. In pratica, sono in grado di mantenere e verificare delle coppie “valori/chiave”.
Puoi pensare ad un contenitore associativo come ad un array che non è obbligato ad usare come chiave un numero intero.
Il contenitore **Map<tipoChiave**, **tipoValore>** permette questo tipo di operazione perfino con l’operatore [], raggiungendo la semplicità dei linguaggi di scripting:

```
map<string, int> numeri;
numeri[“uno”] = 1;
```

In questo caso ho usato un map con chiave a stringa e valore intero, e ho creato una nuova coppia di valori (chiave=”uno”, valore=1) al suo interno.
A seguito di quest’inserimento è possibile recuperare il valore allo stesso modo:

```
cout << numeri[“uno”];
```

```
1
```

In questo caso il programma ha funzionato perché esisteva effettivamente un elemento “uno” nel contenitore.
Potremmo chiederci cosa sarebbe successo in caso contrario.
Quando un elemento viene richiesto tramite operatore [], map lo re-

cupera se esiste, e **ne crea uno nuovo**, **nullo**, **se non esiste**.

```
cout << numeri[“due”]
```

```
0
```

Questo implica che il tipo usato per i valori deve sempre esporre un costruttore senza parametri, per permettere questo tipo di inizializzazione.
Va da sé che in questo modo non si può sapere se un dato elemento esisteva prima della nostra chiamata o meno.
Uno dei modi per risolvere questo problema è usare il metodo **find**(), che restituisce un iteratore all'elemento nel caso in cui l'elemento esista, e un iteratore a **end**() in caso contrario (lo stesso metodo, con il medesimo comportamento, è implementato anche dalle liste).

```
if (numeri.find("tre") != numeri.end())
    cout << "trovato";
```

Per riuscire ad utilizzare l'iteratore passato da find, o da qualsiasi altra funzione (begin, end, etc…), occorre conoscere la reale struttura degli elementi di una map, che è basata su **coppie**, rappresentate dalla classe **pair<chiave, valore>**; i membri **pair::first** e **pair::second** corrispondono rispettivamente alla chiave e al valore degli elementi.

```
map<string, int> numeri;
numeri["uno"] = 1;
numeri["due"] = 2;
numeri["tre"] = 3;

map<string, int>::iterator i = numeri.begin();
cout    << "chiave = " << i->first << endl
```

```
    << "valore = " << i->second << endl;
```

```
chiave = due
valore = 2
```

Il codice riportato qui sopra potrebbe stupire: perché il primo elemento è il due?
La risposta è banale, ma sottolinea un punto importante: perché viene prima in ordine alfabetico (o meglio, lessicografico).
Il contenitore map, infatti, pone gli elementi in ordine crescente, pertanto **è richiesto che la chiave definisca l'operatore <**.
Un'altra funzione importante è make_pair(chiave, valore), che può essere utilizzata per creare delle coppie chiave/valore:

```
numeri.insert(make_pair("quattro", 4));
```

E' interessante anche notare che insert si comporta in modo diverso dall'operatore[]: se un elemento esiste già, **non** modifica quello esistente, ma abbandona l'operazione.

```
map<string, int> numeri;
numeri["uno"] = 0;   //primo assegnamento
numeri["uno"] = 1;   //riassegnamento
numeri.insert(make_pair("uno", 2));  //insert non riassegna
cout << numeri["uno"];
```

```
1
```

5.12.2 ALTRI CONTENITORI ASSOCIATIVI

Sulla stregua di **map**, la libreria standard definisce altri contenitori associativi.
Un **set**, ad esempio, può essere considerato come "una map alla quale mancano i valori", un contenitore associativo in cui l'unica

cosa che conta sono le chiavi, e che è ottimizzato per la loro gestione. Per il resto, il comportamento è del tutto simile a map.
Un **multimap** è un contenitore map in cui le chiavi possono essere duplicate, e che quindi fornisce tutta una serie di funzionalità per gestire le copie (assume importanza, ad esempio, la funzione count(), per contare quante volte si presenti una certa chiave) e disabilita l'operatore []. Analogamente un **multiset** è un contenitore set che permette chiavi multiple.

5.13 PUNTI CRITICI NELL'USO DEI CONTENITORI

Come avrai capito, i contenitori sono tanti e, sebbene espongano funzioni spesso identiche sono tutti diversi a loro modo.
Conoscerli tutti e fino in fondo richiede tempo e spazio, ed esperienza: probabilmente li studierai in modo approfondito sullo Stroustrup [1], via via che ti serviranno. Mi preme, però, metterti in guardia su alcuni punti critici dell'uso dei contenitori, in cui cascano regolarmente tutti i neofiti. Piuttosto che perdermi in una lunga spiegazione circa ciò che avviene dietro le scene, preferisco porti questi paragrafi sotto forma di paternalistici "consigli".

5.13.1 DEFINISCI I COSTRUTTORI E GLI OPERATORI FONDAMENTALI

Come hai visto nel corso di questo capitolo, i contenitori danno molte cose per scontate, riguardo alle tue classi.
In particolare molti pretendono la definizione di alcuni tipi di costruttori e di operatori fondamentali.
Pertanto, cerca sempre di definire:

- **un costruttore di base** (cioè, senza argomenti, o con tutti gli argomenti di default). In questo modo, i contenitori po-

tranno creare dei nuovi elementi su un valore nullo (vedi 5.12.1).
- **un costruttore per copia** (cioè Classe(const Classe&)). Poiché i contenitori operano sempre per copia (vedi 5.12.2)
- **l'operatore di assegnamento**, per lo stesso motivo.
- **l'operatore <**, perché è richiesto dalla stragrande maggioranza dei contenitori e degli algoritmi.
- **l'operatore ==**, perché è richiesto da quei contenitori e quegli algoritmi che non operano per disuguaglianza.

Oltre ad essere utili per i contenitori, tutti questi elementi sono comunque indispensabili per la costruzione di classi robuste.

5.13.2 RICORDA CHE I CONTENITORI AGISCONO PER COPIA

Questo è un punto fondamentale, sia perché implica l'uso di costruttori adatti (vedi paragrafo precedente), sia perché introduce alcuni problemi successivi, sia perché ricordarlo ti evita colossali figuracce davanti al tuo compilatore, a te stesso, e agli altri programmatori.
Ad esempio:

```
vector<int> date;
int oggi = 2006;
date.push_back(oggi);
oggi = 2007;      //non incide su date[0]
cout << date[0]; //date[0] ha ancora il vecchio valore
```

```
2006
```

Il valore dell'elemento date[0] non cambia al variare di "oggi", perché è una semplice copia, e non ha più alcuna relazione con l'originale.

Ricordare che i contenitori "copiano" è utile anche sul fronte dell'ottimizzazione delle prestazioni (l'operazione di copia di un tipo complesso può richiedere molto tempo).

5.13.3 NON DARE PER SCONTATO L'INDIRIZZO DEGLI ELEMENTI

Una delle principali fonti di bug in C++.
Molto spesso alcune classi contengono membri che puntano ad altri oggetti; fin qui, non c'è niente di male: è la semplice relazione di associazione.
I problemi sorgono quando l'oggetto puntato è l'elemento di un contenitore; ciò è rischioso, come dimostra il seguente esempio:

```
vector<string> persone;
persone.push_back("Roberto Allegra");
string* io = &persone[0];
cout << "io sono: " << *io;        //"io sono Roberto Allegra"
persone.push_back("Ignoto Lettore");
cout << "io sono: " << *io;        //probabile crash dell'applicazione
```

Perché la prima volta il cout ha funzionato e quella successiva no?
Semplice: l'aggiunta di te, Ignoto Lettore, ha spinto il contenitore "persone" a cercarsi altra memoria; il contenitore, come spesso accade, non l'ha trovata in loco, e si è trasferito altrove.
La memoria allocata precedentemente è stata rilasciata, e cosi il riferimento "*io" si è corrotto (tipico caso di crisi d'identità digitale).
Sostituisci mentalmente questo semplice esempio con un cocktail di classi interdipendenti, memoria dinamica e distruttori, e potrai facilmente immaginare perché i programmi che puntano direttamente agli elementi di un contenitore incrementale prima o poi vanno in crash.
Le soluzioni sono tre: o si è sicuri che i riferimenti vengano presi solo **alla fine** del processo d'inserimento; o si evita la distruzione de-

gli elementi allocando spazio in anticipo tramite reserve, e controllando che non si esca dai margini prefissati; oppure (soluzione migliore) si adottano altre strategie che non prevedano l'uso di puntatori. Ad esempio, fare riferimento alla posizione di un elemento tramite indice o iteratori, o usare un contenitore associativo.

5.13.4 STA' ATTENTO AI TIPI POLIMORFI!

Uno degli effetti collaterali maggiori del fatto che i contenitori agiscono per copia (vedi 5.12.1), è che non c'è modo di preservare il comportamento polimorfo degli elementi passati.

```
vector<Cane> cani;
cani.push_back(CaneDomestico("pluto"));
cani.push_back(CaneRandagio("ringhio"));
for (vector<Cane>::iterator i = cani.begin(); i != cani.end(); i++)
    i->faiVerso();
```

Bau! Bau!

Dove sono finiti quegli storici latrati "Arf!" e "Grr!" descritti nei primi due capitoli? Si sono persi nella copia effettuata dal contenitore, dal momento che il comportamento polimorfo viene preservato soltanto quando si ha a che fare con riferimenti e puntatori.
Ci sono diversi approcci possibili a questa situazione, ma nessuna "pallottola d'argento" in grado di risolverla banalmente.
Il più ovvio è dichiarare "cani" come vector<Cane*>, ovverosia tramutarlo in un **contenitore di puntatori**.
In questo modo l'elemento memorizzato sarà una copia del puntatore, e preserverà correttamente il comportamento polimorfo.
Peccato che, così facendo, si perda la possibilità di utilizzare la maggior parte degli algoritmi (chi ha interesse ad ordinare dei puntatori?), e si delegittimi il contenitore dal controllo delle proprie strutture (chi si occuperà di distruggere l'elemento?).

Un'altra soluzione è creare una **specializzazione ad-hoc dei contenitori** di tipo vector<T*>: ma ciò richiede tempo, e attenzione, e fatica, il che è proprio ciò che si vuole evitare usando una libreria.
Una soluzione interessante è usare degli **smart pointers** (come quelli forniti dalla libreria Boost [14]), oppure combinare una di queste soluzioni con l'uso di un **garbage collector** [8].

5.14 ALGORITMI

Il C++ fornisce una serie di funzioni template in grado di agire direttamente o indirettamente su array e contenitori standard. Gli **algoritmi** sono dichiarati nell'header **<algorithm>**, sono oltre sessanta, e richiederebbero un libro a parte per una trattazione dettagliata (vedi [1] e [10]).
Si va dagli algoritmi di ordinamento (i vari sort, merge, partition, remove_copy, unique, random_shuffle…), a quelli di iterazione e ricerca (for_each, find, find_if, count, count:_if…), a quelli di riempimento (fill, copy, generate, replace, remove…), alle operazioni su insiemi (set_union, set_difference, set_intersection…), alle combinazioni e permutazioni.
Nel corso di questi paragrafi ne introdurrò qualcuno in modo molto pratico, spiegandone di volta in volta scopo, dinamica e requisiti.
Poiché il C++ implementa gli algoritmi in maniera molto generica, il loro uso può non apparire intuitivo agli occhi del neofita, che solitamente fugge spaventato e si accontenta di "fare da sé" reinventando sistematicamente la ruota. In realtà, dopo la necessaria e sacrosanta fase di studio, gli algoritmi possono rivelarsi una carta vincente nelle mani del programmatore esperto, rendendo la programmazione comoda, snella, coerente e performante.

5.14.1 GLI ALGORITMI "DIETRO LE QUINTE"

Per usare opportunamente gli algoritmi è fondamentale capir bene

come sono realizzati dietro le scene.
La prima caratteristica notevole è che in C++ gli algoritmi non appartengono direttamente ai contenitori, secondo l'approccio OOP, ma sono funzioni generiche che possono essere richiamate indipendentemente dal tipo (array primitivo, list, vector, deque, etc…).
Se hai seguito attentamente quanto espresso negli ultimi due capitoli, avrai certamente capito che gli algoritmi sono **funzioni template**.
Per mantenere la genericità, gli algoritmi operano su **sequenze**, ovverosia accettano sempre come parametro due iteratori che indichino l'inizio e la fine del tratto di contenitore da considerare.
Un esempio pratico è l'algoritmo **void replace()**: qui di seguito è fornita una sua implementazione tipica:

```
template<typename Iteratore, typename Tipo>
    void replace(Iteratore inizio, Iteratore fine,
const Tipo& vecchio, const
Tipo& nuovo)
    {
        //[…] eventuali operazioni per assicurarci che la sequenza
        //indicata sia corretta […]

        for ( ; inizio != fine; ++inizio)
            if (*inizio == vecchio)
                *inizio = nuovo;
    }
```

Prendiamoci un po' di tempo per analizzarlo.
Innanzitutto i primi due parametri rappresentano gli estremi della sequenza (che inizia da "inizio", e finisce **un elemento prima** dell'elemento indicato da "fine"), e appartengono entrambi allo stesso tipo generico "Iteratore".
Gli altri due parametri rappresentano due oggetti dello stesso tipo:

uno è chiamato "vecchio" e uno "nuovo".
Se analizzi un po' il codice della funzione ti verrà facile capire che la funzione **replace** scorre la sequenza alla ricerca di un oggetto uguale a "vecchio", e sostituisce ogni occorrenza trovata con una copia di "nuovo".
Una volta che si è capito tutto ciò, gli algoritmi non sembrano più quei mostri incomprensibili!
Posso assicurarti che l'implementazione degli algoritmi nei compilatori reali non si discosta molto dal codice che abbiamo appena analizzato. Molti compilatori, tuttavia, aggiungono delle macro e delle chiamate a funzione prima di operare, per cercare di rilevare per quanto possibile i casi in cui il programmatore **sbagli nell'indicare la sequenza** – il che avviene più spesso di quanto si creda.
Ad esempio, proviamo a **richiamare l'algoritmo replace**:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Persona {
public:
    string nome;
    string cognome;
    int annoDiNascita;

    Persona(string _nome="", string _cognome="", int
_annoDiNascita=0) : nome(_nome), cognome(_cognome),
annoDiNascita(_annoDiNascita) {};

    bool operator==(const Persona& b) {
    return (
```

```
            nome == b.nome &&
            cognome == b.cognome &&
            annoDiNascita == b.annoDiNascita
    );
    }
};

int main()
{
  vector<Persona> antichi, moderni;
  antichi.push_back(Persona("Marco Tullio",   "Cicerone", -46));
  antichi.push_back(Persona("Caio Giulio",    "Cesare",   -100));
  moderni.push_back(Persona("Alan Mathison",  "Turing",   1912));
  moderni.push_back(Persona("John", "von Neumann",        1903));
  //Sostituisce Cicerone con Seneca
  replace(antichi.begin(), antichi.end(), Persona("Marco Tullio",
 "Cicerone", 26), Persona("Lucio Anneo", "Seneca", -4));

  //Errato: la sequenza mescola antichi e moderni
  replace(antichi.begin(), moderni.end(),
Persona("John", "von Neumann",
1903), Persona("Claude", "Shannon", 1906));
    return 0;
}
```

Prenditi del tempo per studiare questo codice, e capire come funziona generalmente una chiamata ad algoritmo.
Fa' anche molta attenzione all'"errore" presente nella seconda chiamata: il compilatore medio **non è in grado di individuarlo** a priori.
Questo ed altri errori più infidi sono sempre in agguato, pertanto è sempre bene essere sicuri della coerenza dei parametri che si passano come "estremi di sequenza".

5.14.2 ALGORITMI CHE RICHIEDONO FUNZIONI

L’algoritmo **replace**, che abbiamo visto nel precedente paragrafo, fa parte di quell’insieme di funzioni che richiedono in ingresso una o più sequenze, e uno o più valori. Molti algoritmi di ricerca, sostituzione, permutazione e ordinamento fanno parte di questa famiglia.
Si tratta senza dubbio di funzionalità indispensabili (molti programmatori C++ non saprebbero vivere senza il caro e vecchio **find**), ma spesso si rivelano troppo limitati.
Pensa, ad esempio, al semplice caso in cui si voglia modificare il programma visto nel paragrafo precedente, in modo da stampare gli elementi dei contenitori. In questo caso occorrerebbe chiamare la funzione:

```
stampaOgniElementoDelContenitore(antichi.begin(),
antichi.end());
```

Ovviamente, una funzione del genere non esiste (altrimenti dovrebbero esistere anche le altre cinquemila variazioni sul tema).
Invece, esiste la funzione **for_each**, che richiede in ingresso gli estremi della sequenza e **una funzione** che specifichi un’azione da compiere per ciascun elemento della sequenza.
Dietro le quinte, la definizione di for_each può essere la seguente:

```
template<typename Iteratore, typename Funzione>
    Funzione for_each(Iteratore inizio, Iteratore fine, Funzione f)
    {
        for ( ; inizio != fine; ++inizio)
            f(*inizio);
                return f;
    }
```

Come al solito, il funzionamento dell'algoritmo è molto intuitivo, versatile e potente.
Ciò che più ci interessa in questa sede è la chiamata **f(*inizio)**.
Nota come questa scrittura sia assolutamente generica e lasci spazio ad una moltitudine di diverse interpretazioni.
f, ad esempio, potrebbe essere in prima istanza una semplice funzione: in questo caso tutto ciò che viene richiesto è che questa prenda come unico argomento una Persona.
Nel nostro caso, potremmo definirla così:

```
// […] definizione degli include, della classe persona, etc…
[…]

void stampaPersona(const Persona& p) {
    cout << p.nome << " " << p.cognome << "
( " << p.annoDiNascita
<< << ")" << endl;
};

int main()
{
    // […] Definizione dei vettori "antichi" e "moderni"  […]

    for_each(antichi.begin(), antichi.end(), stampaPersona);
    for_each(moderni.begin(), moderni.end(), stampaPersona);

    return 0;
}

```

```
Marco Tullio Cicerone (-46)
Caio Giulio Cesare (-100)
Alan Mathison Turing (1912)
```

```
John von Neumann (1903)
```

Da questo semplice esempio puoi osservare come l'uso di for_each favorisca concisione e riutilizzo, evitando al programmatore di dover scorrere con due cicli for i propri contenitori.

5.14.3 OGGETTI FUNZIONE

Come ho accennato nel paragrafo precedente, il template di for_each è più potente e versatile della semplice "chiamata a funzione": "f" può essere qualunque cosa permetta la scrittura: "f(*inizio)".
Ciò è stato studiato ad arte per superare alcune pesanti limitazioni in cui incorrono le funzioni.
Per illustrare il problema con un semplice esempio, proviamo a modificare il programma in modo che l'output riporti anche un indice numerico crescente, e cioè:

```
Antichi
1) Marco Tullio Cicerone (-46)
2) Caio Giulio Cesare (-100)
Moderni
1) Alan Mathison Turing (1912)
2) John von Neumann (1903)
```

Quest'aggiunta tanto banale mette in crisi il nostro sistema for_each, perché implica che venga utilizzata un'informazione aggiuntiva (l'indice crescente), che non sappiamo proprio dove memorizzare. Neanche l'escamotage rocambolesco di utilizzare una variabile statica o globale viene in nostro soccorso, dal momento che questa dovrebbe essere resettata ad ogni prima chiamata, e noi non disponiamo di alcun mezzo per distinguere una particolare invocazione dalle successive.
Che fare? Rinunciamo al for_each e ci mettiamo a scrivere i cicli a mano?

Questo è proprio il caso di introdurre gli **oggetti funzione**, ovverosia delle semplici classi per le quali sia stato sovraccaricato l'operatore funzione "()", come in quella che segue:

```
class StampaPersona {
public:
    Persona persona;
    int indice;

    StampaPersona() : indice(1) {}

    void operator()(const Persona& p) {
        cout << indice << ") " << p.nome << " " << p.cognome << "
(" << p.annoDiNascita << ")" << endl;
        indice++;
    }
};
```

Quando questa classe viene creata, il costruttore inizializza l'indice sul valore 1. Ogni volta che viene invocata come funzione, l'indice viene incrementato.
Possiamo usarla nel codice in modo naturale:

```
int main()
{
    // […] Definizione di antichi e moderni, etc… […]

    cout << "Antichi" << endl;
    for_each(antichi.begin(), antichi.end(), StampaPersona());
    cout << "Moderni" << endl;
    for_each(moderni.begin(), moderni.end(), StampaPersona());

    system("pause");
```

```
    return 0;
}
```

Il risultato sarà esattamente quello che ci eravamo prefissi.
Ma più che altro è fondamentale che tu capisca bene cos'è successo e non faccia l'errore di confondere lo StampaPersona() che si trova nei for_each con una **chiamata a funzione**.
Perché sarebbe un errore grave. Quello StampaPersona() è una chiamata al **costruttore**: ogni volta, infatti, abbiamo creato un nuovo oggetto di tipo StampaPersona, e per brevità non l'abbiamo inizializzato su una riga separata (puoi farlo, se vuoi).
Le chiamate a funzione, invece, avvengono all'interno del codice template che ti ho mostrato nel paragrafo precedente.
Ciò spiega perché nel secondo for_each l'indice sembra resettarsi magicamente: **si tratta di due oggetti StampaPersona completamente distinti** (che vengono automaticamente distrutti quando non servono più).

5.14.4 OGGETTI FUNZIONE PREDEFINITI

Il bello di usare oggetti funzione è che questi sono elementi riutilizzabili. Il bello di usare una libreria è che questa fornisce molti elementi riutilizzabili.
Risultato del sillogismo: **la libreria fornisce molti oggetti funzione predefiniti**.
Questi si trovano nell'header **<functional>** e seguono i template di base:

```
template<class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

template<class Arg, class Arg2, class Res> struct unary_function {
```

```
    typedef Arg argument_type;
    typedef Arg2 argument_type;
    typedef Res result_type;
};
```

Ciò ha un doppio obiettivo: da una parte si definisce una logica di base coerente per tutti gli oggetti funzione standard; dall'altra si invita il programmatore a sviluppare le proprie estensioni alla libreria in modo da prevedere correttamente il passaggio degli argomenti e la restituzione di un risultato.
La libreria fornisce innanzitutto oggetti funzione che rappresentano le **operazioni aritmetiche** principali (negate, plus, minus, multiplies, divides, modulus), tutti binari ad eccezione del primo.
L'esempio che segue è molto semplice, ed usa l'algoritmo **transform** in congiunzione con **negate** per ottenere un array di numeri negativi, partendo da un array di numeri positivi.

```
int main() {
    int valori[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int valoriNegati[10];

    transform(&valori[0], &valori[10], &valoriNegati[0], negate<int>());

    for (int i=0; i<10; i++)
        cout << valoriNegati[i];

    return 0;
}
0 -1 –2 –3 –4 –5 –6 –7 –8 –9
```

Allo stesso modo, la libreria fornisce un insieme di **predicati**.
Si tratta nella maggior parte dei casi della semplice implementazione degli operatori logici e relazionali (logical_not, logical_and, logical_or,

equal_to, not_equal_to, greater, greater_equal, less, less_equal), tutti binari ad eccezione del primo.
Questi possono trovare un'applicazione in sé e per sé, ma ciò nella pratica avviene di rado, perché i casi del mondo reale sono più complessi di quattro funzioncine da manuale.
Basterebbe chiedersi "come usare un predicato per contare quante persone sono nate prima del 1910", per mettere in crisi tutto il sistema.
Il problema sta nel fatto che **less** (che sarebbe il predicato giusto da usare nell'algoritmo **count_if**) mette in corrispondenza i valori di due sequenze, e non quelli di una sequenza e una costante. Tuttavia questo è proprio il caso più comune.
Per risolvere situazioni come queste la libreria fornisce degli oggetti funzioni noti come **connettori**: bind2nd, ad esempio, lega un predicato ad un secondo argomento, e permette di realizzare facilmente la funzione appena descritta:

```
// […] Definizione di Persona, etc, etc, etc… […]

//definizione dell'operatore <
bool operator< (const Persona& a, const Persona& b) {
    return a.annoDiNascita < b.annoDiNascita;
}

int main() {
    //[…] Definizione di moderni, antichi, bla bla bla… […]
    cout << count_if(moderni.begin(), moderni.end(),
bind2nd(less<Persona>(), Persona(" ", " ", 1910)));
    return 0;
};
```

1 (Cioè Von Neumann. Turing è nato dopo.)

Analogamente, la libreria standard fornisce supporto per altri **adat-**

tatori (come i mai abbastanza lodati adattatori per funzioni membro, quelli per puntatori a funzione e i negatori, in grado di invertire un predicato).
Una spiegazione dettagliata di tutti questi elementi esula largamente dallo scopo del libro (e dallo spazio concesso).

5.14.5 ALGORITMI E INSERITORI

Gli algoritmi della libreria standard del C++ sono il campo che più sbigottisce i neofiti, per il loro aspetto così strano e l'apparente bizantinismo delle loro strutture.
Una volta afferrati i concetti espressi finora, invece, basta una sana dose di studio delle funzioni e dei funtori per diventare produttivi ed evitare fatica.
In quest'ultimo paragrafo voglio aiutarti ancora per un passo a superare un altro possibile scoglio. Immagina l'estensione finale della nostra applicazione, in cui vengono fornite una serie di persone, che vengono scisse in "antichi", se nati prima del 1000, e "moderni", se nati dopo.
Questo è un possibile codice.

```
// […] Definizione delle classi, dichiarazione degli header, etc… […]
// […] Definizione dell'operatore < […]
// […] Definizione dell'operatore >= […]

int main()
{
    Persona persone[4] = { // […] Elenco delle persone […]
    };

    vector<Persona> antichi(4), moderni(4);

    remove_copy_if(&persone[0], &persone[4], antichi.begin(),
bind2nd(greater_equal<Persona>(), Persona("", "", 1500)));
```

```
    remove_copy_if(&persone[0], &persone[4], moderni.begin(),
bind2nd(less<Persona>(), Persona(" ", " ", 1500)));
    // […] Stampa delle persone antiche e moderne […]
    return 0;
}
```

Per inciso, qui ho utilizzato l'algoritmo **remove_copy_if**, che inserisce nell'iteratore passato come terzo parametro una lista contenente tutti gli elementi **eccetto** quelli che soddisfano un predicato. Dopo una prima analisi (sì, d'accordo: il fatto che non sia sopravvissuto il più diretto "**copy_if**" è effettivamente un bizantinismo), il giro logico dovrebbe apparire chiaro.
E allora perché mai l'output è questo?

```
Antichi
1) Marco Tullio Cicerone (-46)
2) Caio Giulio Cesare (-100)
3) (0)
4) (0)
Moderni
1) Alan Mathison Turing (1912)
2) John von Neumann (1903)
3) (0)
4) (0)
```

La risposta è semplice se guardi come ho dichiarato i vettori antichi() e moderni(): ho creato già in partenza quattro elementi vuoti. Ciò è necessario, perché gli algoritmi non si occupano di aggiungere o togliere elementi ai contenitori, **ma solo di sostituire e spostare**. Quindi, per evitare il rischio di crash per overflow, sono stato costretto a "prenotare" la dimensione massima possibile.
Per rimediare la libreria standard fornisce due vie.
La prima è aperta dal fatto che le funzioni che generano nuove se-

quenze (come remove_copy_if) ritornano un iteratore all'ultimo elemento valido, che può essere usato per ridimensionare il contenitore. Ciò è semplice nel caso dei vettori:

```
antichi.resize(remove_copy_if(&persone[0],
&persone[4], antichi.begin(),
bind2nd(greater_equal<Persona>(), Persona(" ", " ", 1500))) –
antichi.begin());
```

L'esempio riportato qui sopra è valido e corretto, ma solo per i vettori: altri contenitori memorizzano gli elementi in posizioni non contigue, pertanto per essi non ha senso un'operazione di differenza. Inoltre questa scrittura non risolve la necessità della "prenotazione di spazio" e risulta pacchiana e ridondante.
La seconda via implica il ricorso a delle geniali specializzazioni di iteratori chiamate **inseritori**, che fanno in modo di aumentare le dimensioni del contenitore automaticamente.
Tutto ciò che viene richiesto al programmatore è di includere l'header **<iterator>**, e richiamare nell'argomento d'uscita la funzione **back_inserter**(**contenitore**) o **front_inserter**(**contenitore**) a seconda che si voglia rispettivamente aggiungere in testa, o in coda.

```
vector<Persona> antichi, moderni;

remove_copy_if(&persone[0], &persone[4], back_inserter(antichi),
bind2nd(greater_equal<Persona>(), Persona(" ", " ", 1500)));
remove_copy_if(&persone[0], &persone[4], back_inserter(moderni),
 bind2nd(less<Persona>(), Persona(" ", " ", 1500)));
```

Questo semplice accorgimento risolve ogni problema.
A questo punto non hai più scuse che ti trattengano dallo sviscerare lo Stroustrup o un altro libro di riferimento, alla ricerca dell' "AlgoritmoCheFaAlCasoTuo"!

STRINGHE E CANALI

Durante il corso di questo libro e di “Imparare C++” ho fatto spesso ricorso alla classe **string** e agli stream predefiniti (cin e cout, per esempio). Ciò dimostra che è possibile usare questi componenti senza conoscerne l’esatto funzionamento, un po’ come si faceva in C con i vecchi “printf” e “scanf”. In questo capitolo esamineremo questi aspetti del C++ in maniera più approfondita, in modo da capirne le possibilità e i limiti.

6.1 CHAR_TRAITS

Gli stream e le stringhe sono accomunati dal fatto che entrambi operano su “sequenze di caratteri”, il che è considerato dai novizi sinonimo di **“array di char”**.

Questa visione del problema si frantuma ben presto, quando viene posta di fronte a termini come “Unicode” (caratteri di 16 bit), “Estensioni alternative dell’ASCII” (latin1 e latin2), e alla gran varietà di caratteri, alfabeti, mappature e codepages presenti sul pianeta.

Per questo il C++ definisce un carattere attraverso una struttura template che può essere specializzata: **char_traits<>**.

Un char_traits espone una serie di tipi che rappresentano un carattere, e di metodi statici che ne gestiscono le operazioni fondamentali. Ad esempio:

```
char_traits<char>::to_char_type(64);         //@
char_traits<char>::to_int_type('@') ;        //64
char_traits<char>::eq('@', '@') ;  //vero (@ = @)
char_traits<char>::lt('A', 'C') ;            //vero ('A' < 'C') ;
```

Tutte queste operazioni risultano banali per un char, ma acquistano un’importanza fondamentale per i caratteri di altro tipo, come **wchar_t** (caratteri estesi).

Char_traits<> espone anche una serie di metodi statici per la gestione

e il confronto di array di caratteri. Qui ti presento degli esempi:

```
char stringa[] = "Ciao!";
char_traits<char>::length(stringa);          //5

//trova la prima occorrenza di "i" nelle prime tre lettere
const char* lettera = char_traits<char>::find(stringa, 3, 'i');
//il valore restituito è un puntatore al carattere trovato
(lettera == &stringa[1]); //vero

char copia[6];
//copia stringa in stringa2
char_traits<char>::copy(copia, stringa, 6); //copia 6 lettere
//verifica se stringa1==copia
char_traits<char>::compare(stringa1, copia, 6) ;
//0: sono uguali
```

Queste funzioni evitano il ricorso alle funzioni C, come strlen(length), strcpy(copy), o strcmp(compare), ed altre ancora.

6.2 BASIC_STRING

Char_traits definisce già un buon insieme di operazioni per la gestione delle stringhe, seppure a un livello piuttosto basso e simile al C. Il C++ offre una classe template, **basic_string<T>**, per la gestione ad alto livello delle sequenze del **char_traits<T>** corrispondente. Ad esempio, basic_string<char> implica l'uso (e la definizione) di char_traits<char>.
Dal momento che sarebbe molto noioso dover specificare ogni volta il nome completo, la libreria C++ definisce automaticamente dei typedef simili a questi:

```
typedef basic_string<char> string;
```

```
typedef basic_string<wchar_t> wstring;
```

Così è possibile utilizzare **string** (o un’altra istanza di basic_string) come se fosse un tipo di dato o una classe base. In realtà, come abbiamo visto, si avvicina molto di più ai contenitori della libreria standard: può essere vista come un vector di char, con metodi molto più ottimizzati e funzionalità specifiche, che illustrerò in maniera sommaria nei paragrafi successivi.

6.2.1 COSTRUZIONE

Innanzitutto i **costruttori** permettono di inizializzare una stringa in diversi modi: vuota; a partire da una stringa C letterale; specificando il numero di elementi; inizializzando un carattere per un certo numero di volte; copiando un’altra stringa; copiando parte di un’altra stringa, indicando il punto di inizio e il numero di elementi, oppure gli estremi tramite due iteratori.

```
string str;     //costruttore di base
string str2("C++!");  //costruttore con stringa letterale
string str3(20);    //costruttore con numero di elementi
string str4(10, “*”);   //str4 = “**********”
string str5(str4);  //costruttore per copia
string str6(str2, 0, 2); //str2[0] e str2[0,1]
string str7(str2, str.begin(), str.end()); //str2 = C++!
```

6.2.2 DIMENSIONE E CONFRONTO

È sempre possibile ottenere le informazioni tipiche di un contenitore (i tipi descrittori, gli iteratori), e i metodi relativi alla **dimensione**. Come sinonimo di size() è possibile usare **length()**.

```
string str = "StringaDiVentiseiCaratteri";

  str.length() == 26;                //vero!
```

Si possono eseguire **confronti** fra stringhe o fra stringhe e array di caratteri, secondo l'ordine lessicografico (o specificandone un altro, volendo) con gli operatori standard, evitando così le criptiche funzioni di compare.

```
string str1 = “giabim”;
string str2 = “roberto”;

str1 ==  “roberto”;    //vero
str1 != str2; //vero
str1 < str2;  //vero: g < r
```

6.2.3 CONCATENAZIONE, AGGIUNTA E INSERIMENTO

Basic_string definisce l'operatore + per la concatenazione fra stringhe o fra stringhe e array di caratteri.
Coerentemente, è possibile utilizzare l'operatore di incremento per aggiungere una stringa o un array di caratteri ad una basic_string.

```
string str = string("Pian") + "ta";               //str = "Pianta";
str += "la";                    //str = "Piantala";
```

Per l'inserimento di una stringa o di un array di caratteri in una data posizione, ci si può avvalere della funzione **insert(posizione, stringa)**.

```
str.insert(6, "te");  //str = "Piantatela";
```

6.2.4 RICERCA

Basic_string<> fornisce molte funzioni per la ricerca della prima occorrenza di una stringa in un'altra (find) e dell'ultima (rfind); inoltre è possibile specificare un insieme di caratteri della quale si vuole ricercare la prima occorrenza (**find_first_of**), l'ultima (**find_last_of**)

oppure la prima parte della stringa che non occorre in un'altra, partendo dall'inizio (**find_first_not_of**) o dalla fine (**find_last_not_of**). Ciò che viene restituito è un indice relativo alla posizione dell'occorrenza trovata

```
string str = "barbasso";
str.find("ba");             //0. Infatti, str[0] = 'b' e str[1] = 'a'
str.rfind("ba");            //3. Infatti, str[3] = 'b' e str[4] = 'a'
str.find_first_of("ao");       //1. Infatti, str[1] = 'a'
str.find_last_of("ao");        //7. Infatti, str[7] = 'o'
str.find_first_not_of("ao");          //0. Infatti, str[0] != 'a' e str[0] != 'o'
str.find_last_not_of("ao");           //6. Infatti, str[6] != 'a' e str[6] != 'o'
str.find("moscardo"); //npos
```

Nell'ultima istruzione è stata ricercata una sottostringa non esistente in str. In questo caso viene restituita la costante statica **npos**, che indica un valore errato. Usare npos come indice porta ad un'eccezione di tipo **range_error**.

```
str[str.find("moscardo")] = 'a';                    //range_error
```

6.2.5 SOTTOSTRINGHE

L'ultimo assegnamento, oltre ad essere errato, è interessante per altri versi: talvolta capita di voler referenziare una parte di una stringa (non solo una lettera: un'intera sottostringa) e cambiarla, o cancellarla.
Il metodo **replace** (**inizio**, **ncaratteri**, **sostituzione**) offre un primo aiuto: cancella i caratteri della stringa dal punto indicato in "inizio", nella quantità indicata da "ncaratteri", e inserisce all'interno la stringa indicata in "sostituzione".

```
string str = "vulneraria";
str.replace(str.find("aria"), 4, "abile");          //str = "vulnerabile"
```

Il metodo **erase** (**inizio**, **nCaratteri**) è una versione più efficiente di replace(inizio, ncaratteri, “ ”).

```
str.empty(0, 6); //str = “aria”
```

Per ottenere una sottostringa “in lettura”, invece, è possibile utilizzare la funzione **substr**(**inizio**, **nCaratteri**).

```
cout << str.substr(6,4);
```

```
aria
```

6.2.6 CONVERSIONE AD ARRAY DI CARATTERI

Difficilmente la classe basic_string sarà rappresentata attraverso un semplice array di carattere a terminatore nullo: implementazioni efficienti delle stringhe possono prevedere buffer separati, strutture di controllo, e altro ancora. Tuttavia, può succedere di voler accedere ad una versione primitiva dei dati, per esempio per usare una libreria C.In questo caso è possibile utilizzare le due **funzioni data**(che restituisce un array di caratteri) e **c_str**(che restituisce un array di caratteri con terminatore nullo). La cosa positiva è che queste stringhe sono già gestite dalla classe, pertanto non c’è bisogno di eliminarle con delete[] (anzi, è vietato).

```
string str = “210.5”;
double d = atof(str.c_str());
```

6.3 STREAM DI INPUT E DI OUTPUT

Il C++ raggruppa tutti quei casi in cui qualcosa debba essere convertito da o verso un flusso di dati, grazie al meccanismo degli stream. Uno **stream** (o canale) è generalmente un canale di dati, in ingresso

(istream) o in uscita (ostream).
Qui approfondirò alcuni aspetti degli stream in maniera molto pratica, rimandando alla lettura di [1] l'analisi di molte altre funzioni.

6.3.1 OSTREAM

Un stream di output è un'istanza della classe template **basic_ostream<T>**, laddove T rappresenta il tipo di caratteri (espresso dal char_traits corrispondente) che sarà dato in uscita. Analogamente a char_traits, esistono dei typedef per semplificare la dichiarazione degli stream più comuni, ad esempio:

```
typedef basic_ostream<char> ostream;

typedef basic_ostream<wchar_t> wostream;
```

L'operazione tipica di un ostream è l'uso dell'operatore <<, che **inserisce** un ostream in un altro e restituisce lo stream stesso, in maniera che si possano scrivere correttamente inserimenti multipli.
Ad esempio:

```
cout << 1 << 0 ;
```

In questo caso, il compilatore interpreterà la chiamata come :

```
(cout.operator<<(1)).operator<<(0);
```

La classe definisce solo un ridotto insieme di sovraccaricamenti per l'operatore << (per i tipi di dati primitivi, e per altri stream).
Tutti gli altri (inclusi quelli definiti dall'utente) vengono sovraccaricati all'esterno, ad esempio nel caso di una stringa in stile C il prototipo sarà questo:

```
ostream& operator<< (ostream& os, const char* str );
```

Ciò non implica, comunque, cambiamenti sostanziali nella programmazione "ad alto livello": chi usa gli stream può sempre utilizzare, ad esempio, una chiamata di questo tipo:

```
cout << "Rob" << "erto";
```

senza dover necessariamente rendersi conto che, dietro le scene, il compilatore la interpreta come :

```
operator<<(operator<<(cout, "Rob"), "erto");
```

6.3.2 ISTREAM

Analogamente a **basic_ostream<>** il C++ fornisce la classe basic_istream<> per gli stream di ingresso, ovverosia quelli in cui una sequenza di caratteri viene convertita in un dato di altro tipo.
Altrettanto analogamente, sono definiti i due typedef:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
```

Mentre ostream permette l'inserimento dei dati, istream ne consente l'estrazione, attraverso l'operatore >>. Ad esempio:

```
int x;
cin >> x;     //estrae da cin in x
```

È fondamentale sapere che gli istream non estraggono **tutto** il contenuto dello stream, ma si fermano quando incontrano uno spazio bianco (anche se saltano quelli iniziali).
Quindi in un caso del genere:

```
int x = 0, y = 0;
cin >> x ;
```

```
cin >> y ;
```

Se alla prima richiesta l'utente scriverà: "2 3", x varrà 2 e y 3.
Colgo quest'occasione, peraltro, per farti notare che il "**cin** via tastiera" non implica sempre una richiesta di dati all'utente: solo quando i dati nel buffer finiscono, il programma si arresta per chiederne altri.
Non sempre le operazioni di estrazione vanno a buon fine.
Ad esempio, nel caso in cui l'utente inserisca "a b" all'esecuzione del codice precedente, le operazioni non saranno eseguite, e x e y manterranno il loro valore (0).
Uno stream (e pertanto anche lo stream derivato da un inserimento) può essere usato come una variabile booleana, per capire se si trova in uno stato buono, o non valido.

```
int x;
if (cin >> x) {

    cout << "hai inserito un numero valido";
}
else
{
    //gestisci errore
}
```

Per l'estrazione in un carattere si può usare la funzione **get**(**destinazione**), e per una stringa **getline**(***destinazione, caratteri, terminatore**).

```
char x;

cin.get(x);
char stringa[40];
cin.getline(stringa, 40, '\n');
```

6.4 STREAM PREDEFINITI

La libreria standard fornisce in <iostream> una serie di stream predefiniti; niente di nuovo: abbiamo già usato per tutto il corso del libro gli stream **cout** e **cin**.
L'elenco completo è fornito in tabella 5.11.

Stream	Tipo	Carattere	Scopo
cout	uscita	char	Canale predefinito per l'uscita
cerr	uscita	char	Log degli errori non bufferizzato
clog	uscita	char	Log degli errori bufferizzato
wcout	uscita	wchar_t	Canale predefinito per l'uscita
wcerr	uscita	wchar_t	Log degli errori non bufferizzato
wclog	uscita	wchar_t	Log degli errori bufferizzato
cin	ingresso	char	Canale predefinito per l'ingresso
wcin	ingresso	wchar_t	Canale predefinito per l'ingresso

Tabella 5.11: Elenco degli stream predefiniti

Normalmente questi stream vengono preimpostati dal sistema, e non c'è bisogno di preoccuparsene: occorre comunque tenere a mente che ingressi e uscite possono essere ridirezionate.

6.5 STRINGSTREAM

Ostream e istream non sono le uniche classi che operano su string. **Basic_stringstream<T>**, ad esempio, è una derivazione di basic_stream<T>, che permette di associare uno stream ad una stringa.
L'header <sstream> stabisce anche i seguenti typedef (e i loro corrispettivi estesi):

```
typedef basic_istringstream<char> istringstream;        //Input
typedef basic_ostringstream<char> ostringstream;        //Output
typedef basic_stringstream<char> stringstream;          //Input / Output
```

Uno stringstream è uno stream che rappresenta l'astrazione di una stringa, in grado di aumentare la sua dimensione in modo dinamico. È possibile usare uno stringstream come se si scrivesse su cout, e richiamare la funzione str() per ottenere una string.
Grazie a stringstream si può trasformare un numero in una stringa (evitando così il ricorso a funzioni come itoa, ftoa, eccetera), o la scrittura avanzata su una stringa con manipolatori e formattatori.

```
ostringstream stringa;
stringa << 1984 << " e' stato scritto nel " << 1948;
cout << stringa.str(); //1984 è stato scritto nel 1948
```

Allo stesso modo, è possibile utilizzare l'estrazione per trasformare una stringa in un valore numerico (evitando così il ricorso a funzioni come atoi, atof, eccetera).

```
string stringa("1984");
istringstream str(stringa);

int numero;
str >> numero;

cout << numero - 36;//1948
```

6.6 FSTREAM

Basic_fstream<T> è una derivazione di basic_stream<T> che permette di associare un file ad una stringa.
L'header <fstream> stabilisce anche i seguenti typedef (e i loro corrispettivi estesi):

```
typedef basic_ifstream<char> ifstream;                    //Input
typedef basic_ofstream<char> ofstream;       //Output
```

```
typedef basic_fstream<char> fstream;                    //Input/Output
```

Un fstream rappresenta l'astrazione di un file, in scrittura (ofstream) per accedere a un file e scrivervi, mediante inserimento:

```
ofstream f("testo.txt");
f << 33 << " trentini entrarono in Trento…";
```

Da questo breve estratto si possono dedurre molte cose: innanzitutto che il file da aprire può essere indicato nel costruttore. È anche possibile indicare, in un secondo argomento, la modalità d'accesso, scegliendo (o combinando) fra: **in** (lettura), **out** (scrittura), **app** (aggiunta), **ate** (partenza dalla fine del file), **binary** (in binario) e **trunc** (troncando a zero). Infine, nota anche il fatto che la classe si prende cura automaticamente di aprire e chiudere il file. Qualora queste operazioni debbano essere forzate (ad esempio, per aprire files diversi, o evitare accessi concorrenti), è possibile usare le funzioni **open()** e **close()**.
In lettura (istream) è possibile accedere a un file e leggervi, mediante inserimento:

```
ifstream f(testo.txt);
int numeroTrentini;
f >> numeroTrentini;

cout << "Ultime notizie: i trentini sono " << numeroTrentini; //33
```

6.7 FORMATTAZIONE E MANIPOLATORI

In ogni stream è definita la stessa serie di costanti statiche di tipo **ios_base::fmtflags**, che rappresentano un particolare set di formattazioni che viene impiegato nel trattamento dei dati.

Attraverso la funzione flags() è possibile ottenere una variabile che rappresenta le loro impostazioni al momento della chiamata, la funzione **flags**(**nuoviIndicatori**) permette di reimpostarli, e la funzione **setf**(**nuovoIndicatore, maschera**) permette di aggiungere un indicatore di tipo "maschera" a quelli già impostati.
Questo sistema permette di "salvare" lo stato del canale, applicare dei cambiamenti alla sua formattazione, e poi ripristinarlo usando la copia, in modo simile:

```
ios_base::fmtflags impostazioni = cout.flags();
cout.setf(ios_base::hex, ios_base::basefield);
cout << 256 << ' ';
cout.flags(impostazioni) ;
cout << 256;
```

```
100
256
```

In questo caso ho utilizzato la costante **ios_base::hex**, per indicare a cout di trattare i numeri successivi come esadecimale.
Analogamente, si può usare **ios_base::dec**, per indicare un numero decimale, e **ios_base::oct**, per un ottale. Il secondo argomento "maschera" è necessario per stabilire che si vuole indicare un'operazione su un numero intero (**ios_base::basefield**).
Analogamente, è possibile usare la maschera **ios_base::floatfield**, per i numeri in virgola mobile, rappresentabili con formato normale (**0**), scientifico (**ios_base::scientific**) o fisso (**ios_base::floatfield**).

```
cout.setf(ios_base::scientific, ios_base::floatfield);
cout << 3.141592653;
```

```
3.141593e+000
```

Un altro uso tipico delle costanti di formattazione è specificare l'allineamento di un campo. Gli stream, infatti, espongono le funzioni **width** e **fill** per impostare un sistema simile a quello delle tabulazioni degli editor di testo. **Width** specifica di quanti caratteri (minimo) deve essere il testo del successivo inserimento, e **fill** il riempimento da usare nel caso in cui la lunghezza del testo scritto sia inferiore a width(). Ad esempio:

```
cout.width(10);
cout.fill('*');
cout << "Roberto";
cout.width(10);
cout.fill('*');
cout << "Allegra";
```

```
***Roberto***Allegra
```

È possibile passare delle impostazioni di formattazione assieme alla maschera **ios_base::adjustfield**, per indicare il comportamento standard del riempimento.
Ad esempio, **ios_base::right** inserisce il testo a destra del riempimento (come nell'esempio), e **ios_base::left** a destra.

```
cout.width(10);
cout.fill('*');
cout.setf(ios_base::left, ios_base::adjustfield);
cout << "Giabim";
```

```
Giabim****
```

Usare i metodi di formattazione può essere molto fastidioso, perché porta alla scrittura di molte righe di codice che generano confusione nel lettore, e appesantiscono inutilmente il sorgente. Inoltre, al-

cune funzioni, come flushing del canale e l'impostazione delle dimensioni dei campi con width() e fill(), devono essere eseguite ad ogni operazione. Per questo, il C++ implementa un sistema di **manipolatori**: si tratta di funzioni che possono essere richiamate direttamente nel corso dell'operazione di inserimento (o di estrazione), grazie a dei sovraccaricamenti apposti di operator<< (o di operator >>), che permettono di richiamare un puntatore a funzione come argomento. Il risultato è che è possibile scrivere qualcosa del genere:

```
cout << hex << 256;
```

Nota come la scrittura sia molto più comprensibile, comoda, e compatta; **hex** è una funzione globale che richiama correttamente la formattazione dello stream su ios_base::hex.
Esistono manipolatori per ogni formattazione del canale, e altri ancora, come **flush**, per svuotare il canale, **endl**, per aggiungere un ritorno a capo e richiamare flush, **ends**, per aggiungere un carattere nullo e richiamre flush, e così via. Con un meccanismo analogo, l'header <iomanip> definisce dei manipolatori che accettano argomenti, come **setw**(che richiama width), **setfill**(che richiama fill), **setprecision**(che richiama precision, e indica il numero di cifre dopo la virgola da mostrare nei decimali), eccetera:

```
cout << left;
cout << setw(10) << "Nome" << setw(10)
<< "Cognome" << endl;
cout << setw(20) << setfill('-') << "" << endl;
cout << setw(10) << setfill(' ') << "Roberto" <<
    setw(10) << "Allegra" << endl;
```

```
Nome      Cognome
--------------------
Roberto   Allegra
```

Per un elenco dei manipolatori, degli argomenti, dei formati ed altre informazioni su stringhe e stream, puoi consultare i riferimenti in bibliografia.

6.8 CONCLUSIONI

Con questa panoramica essenziale della sterminata libreria standard, siamo giunti alla fine del libro. Se sei arrivato fin qui, avendo assimilato capitolo dopo capitolo, avrai una buona base di C++, che non aspetta altro che essere consolidata dalla pratica continua.
Se sei nuovo della programmazione, il mio consiglio è: scrivi in C++! Impara a gestire nuove situazioni e la miriade di problematiche correlate: uno studio attento dei pochi ma essenziali riferimenti che ho fornito in bibliografia dovrebbe costituire una base molto solida per qualunque programmatore. Insomma, ora che siamo arrivati alla fine, posso augurarti un buon inizio nella via del C++!

BIBLIOGRAFIA E SITOGRAFIA

Mettendo assieme dieci libri simili a questo e scrivendo con un carattere molto minuto, forse si riuscirebbe a stendere una bibliografia soddisfacente per il C++.
Dovendomi occupare di altre cose, in questo volume, ho pensato di restringere la bibliografia al minimo assoluto.
Questi riferimenti non devono essere intesi come delle **possibilità** con cui ampliare i propri orizzonti, ma come "quei libri di base senza i quali non ci si può definire programmatori C++, e in alcuni casi neanche programmatori".
Salvo riferimenti precisi ad articoli e libri irripetibili (ad esempio: [1], [4] e [8]), tutti i titoli sono pienamente sostituibili con decine di altri degni equivalenti: da questo punto di vista i volumi consigliati sono puramente indicativi.

[1] **"Programmare in C++, terza edizione"**, Bjarne Stroustrup, Addison-Wesley.

[2] **"The design and evolution of C++"**, Bjarne Stroustrup, Addison-Wesley.

[3] **"Il linguaggio C"**, Kernighan Brian W.; Ritchie Dennis M., Pearson Education.

[4] **"Goto statement considered harmful"**, Edsger W. Dijkstra. Letter to Communications of the ACM (CACM) vol. 11 no. 3, March 1968, pp. 147-148.

[5]**"Algoritmi in C++"**, Robert Sedgewick Pearson Education Italia.

[6] **"A Garbage Collector for C++"**,
http://www.hpl.hp.com/personal/Hans_Boehm/gc/

[7] **“Applying Uml and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3nd Edition”** Craig Larman. Prentice Hall 2005

[8] **“Design patterns”**: Gamma, E., Helm, R., Johnson, R. e Vlissides, J (altrimenti noti come la Gang Of Four). Pearson Education Italia.

[9] **“Ingegneria del software”**, Sommerville Ian. Pearson Education Italia.

[10] **“The C++ Standard Library: A Tutorial and Reference”**, Nicolai M Josuttis, Addison-Wesley.

I riferimenti che seguono, invece, non sono indispensabili per saper programmare, ma possono risultare molto utili ai lettori più smaliziati per la comprensione di alcuni aspetti del C++ più profondi o marginali, e per implementare soluzioni avanzate.

[11] **“Advanced Compiler Design and Implementation”**, Steven Muchnick, Academic press.

[12] **“Extreme Programming Explained: Embrace Change”**, Kent Beck, Addison-Wesley Professional.

[13] **“Generic Programming and the STL: Using and Extending the C++ Standard Template Library”**, Matthew H. Austern, Addison-Wesley Professional.

[14] **“Libreria Boost C++”**, http://www.boost.org